Contributions to Formal Specification and Modular Verification of Parallel and Sequential Software

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Alan David Weide, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2021

Dissertation Committee:

Paolo A.G. Sivilotti, Advisor

Michael Bond

Murali Sitaraman

Neelam Soundarajan

Abstract

Modular verification of parallel and concurrent software built from reusable data abstractions is a challenging problem. Reasoning about sequential software can be modularized using the specifications of data abstractions, but the need to consider implementation details complicates reasoning about parallel execution.

Addressing this challenge requires advancing the state of the art in several ways, beginning with a theoretical foundation. The A/P Calculus for describing the effects of program actions is developed in this dissertation to enable sound modular reasoning about parallel programs with non-interfering parallel sections of operation calls on abstract data types. Building on the calculus and a programming language with clean semantics, a methodology for designing decomposable data abstractions is presented to produce fork-join parallel programs that are manifestly data race free and readily amenable to modular reasoning. A new specification construct, the non-interference contract, is proposed to enhance the specification of data abstractions to hide implementation details and yet facilitate modular reasoning about parallel programs that share objects among processes.

As a key first step to transition these results to practice, this dissertation describes Clean++, a discipline for writing software in C++ that leverages move semantics to make ownership transfer the primary data movement operation (as opposed to either deep or shallow copying) and produce programs that are amenable to formal verification with only minimal scaffolding related to pointer manipulation.

For my parents, Bruce and Jane Weide,

for my wife, Kayla Sands,

and for all of my friends and family who have supported me in this endeavor.

Acknowledgments

First and foremost, not a single word of this dissertation would have been written without the unwavering support and continuous encouragement from my advisor, Dr. Paul Sivilotti, and *de facto* co-advisor, Dr. Murali Sitaraman. To the two of you, your apparently infinite patience was instrumental in my seeing this through to the end, and I cannot thank you enough for it. I must also, of course, acknowledge your intellectual guidance through the twists, turns, forks, and joins of Ph.D. research that helped me tremendously in collecting my thoughts cohesively into a story that anybody but myself would be able to read. Without your superhuman commitement to me and your other students, I would have been lost several years ago.

I would also like to thank the members of my committee not only for their time in serving on my committee, but also for shaping my interests throughout my entire collegiate career at Ohio State. To Dr. Neelam Soundarajan, thank you for supporting my entrepreneurial side as the faculty advisor for the NEWPATH group. Without that support I would not have had the courage and knowledge to pursue Accelowait (even though in the end it did not make me rich). During my time in graduate school, your healthy skepticism at my various presentations in the RSRG seminars helped hone this research. To Dr. Mike Bond, your seminar in the fall of 2015 was one of the first graduate-level courses I took. The exposure to research in concurrency from that course convinced me to pursue a Ph.D.

I would be remiss not to thank all of the members of RSRG at Ohio State, Clemson, and elsewhere both for their engaging conversations and presentations that I was privileged to attend on a weekly basis and for their detailed, thoughtful, and challenging feedback to my own research. In particular, to Dr. Bill Ogden, your detail-oriented attitude toward my work helped sharpen its edges, to Dr. Diego Zaccai, our conversations about the minutae of my work were always clarifying, and to Dr. Wayne Heym, your singular kindness and optimism was a light in every RSRG seminar, but most importantly when I was doubting myself.

To The Ohio State University, I know that without the Buckeye community's spirit, enthusiasm, and fostering of curiosity, I would never have begun graduate school in the first place. To The Best Damn Band In The Land (and especially F-Row), thank you for providing an outlet for my creativity and a constant source of youthful energy even as we aged.

Finally, I think it is impossible to express the enormity of the gratitude I have toward my family. To my sister, Lauren, thank you for always pretending you were interested in my research, even though I knew you'd rather talk about anything else. To my mother, Jane, thank you for asking how much work I got done each day we had our family dinners. Without you holding me accountable, I would probably have taken an extra year to finish this dissertation. To my father, Bruce, thank you for dually serving as a loving, supportive father and as a detatched colleague. Thank you for being a sounding board for my research ideas and for helping me to navigate not only adult life in general, but the life of an academic more specifically. To my wife, Kayla, you have been my rock for these past several years despite living on a different continent for most of them. You have been both my cheerleader when I achieved a milestone and a shoulder to cry on when I missed one. I am eternally indebted to you for your unending support, patience, and love. Without you, this would not have been possible. To our dog, Roslin, thank you for always being happy to see me. Without you, this would have been possible, but it would have been a much quieter journey.

Alan D. Weide

Columbus, Ohio

October 14, 2021

Vita

2015	B.S. Computer Science and Engineering, The Ohio State University
2020	M.S. Computer Science and Engineering, The Ohio State University
2015–present	Graduate Teaching Associate, The Ohio State University

Publications

Research Publications

Alan Weide, Paolo A.G. Sivilotti, and Murali Sitaraman. "An Array Abstraction to Amortize Reasoning about Concurrent Client Code". In: *Intelligent Computing: Proceedings of the 2021 Copmuting Conference*, vol. 1, pp 346–362, London, UK, July, 2021.

Alan Weide. "Reasoning Challenges of Data Abstraction and Aliasing in Concurrent Programs". In: *SIGSOFT Software Engineering Notes*, vol. 43, issue 3, pp 18, July 2018.

Alan Weide. "Enabling Modular Verification of Concurrent Programs". In: *European Conference on Object-Oriented Programming Doctoral Symposium*, Barcelona, Spain, June 2017.

Alan Weide, Paolo A.G. Sivilotti, and Murali Sitaraman. "Enabling Modular Verification with Abstract Interference Specifications for a Concurrent Queue". In: *Verified Software. Theories, Tools, and Experiments*, pp 119–128, Toronto, Canada, July, 2016.

Alan Weide. "Intermediate Models for Expressing Parallelization Opportunities with Abstract Data Types" In: *RESOLVE 2016*, Columbus, OH, June 2016.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Software Engineering	Prof. Paolo A.G. Sivilotti
Theory and Algorithms	Prof. Kenneth J. Supowit
Computer Graphics	Prof. Han-Wei Shen

Table of Contents

List of Source Listings

List of Figures

Chapter 1: Introduction

Parallelism is one of an engineer's best tools to improve the performance of software; however, parallel programming brings a set of challenges unique from sequential programming. Therefore, as computing hardware becomes ever more parallel and parallelism is exploited to improve the performance of mission- and safety-critical software, the importance of making sure parallel software is correct increases. When multiple processes are executed in parallel in a shared-memory environment, they have the possibility of interfering (*e.g.*, they might concurrently write data to a single memory location). Interfering processes are said to exhibit a data race, and their combined behavior is inherently nondeterministic. However, when several processes execute in parallel but *do not* exhibit any data races, their execution is inherently *deterministic*—it is these kinds of systems that garner the majority of attention in this work.

Complicating reasoning about concurrent programs is the fact that most parallel and concurrent software at present is written with low-level software patterns, such as explicit locks or semaphores, and does not normally involve rich abstractions to aid in reasoning about the behavior of programs. One reason that this programming style is popular is that in order to guarantee the absence of data races, it is necessary to have sufficient knowledge of the implementation of a software component. A key contribution of this dissertation is showing that this knowledge can be attained without sacrificing rich abstraction.

In sequential software, abstraction is commonly used to ease reasoning (and therefore verification) by increasing modularity. Unfortunately, guaranteeing simultaneously the modularity of the verification process and the independence of concurrent threads is complicated by two key challenges. The first is how to account for data sharing between threads such as through aliased references, and the second is how to guarantee safe (*i.e.*, deterministic) parallel execution of operations on an object without violating abstraction.

In total, there are four main contributions of this dissertation.

1. A calculus to serve as the foundation for provably sound modular reasoning about parallel programs

2. A methodology for designing data abstractions that permit safe parallelism without the need for additional specification constructs within a language that does not permit aliased references

3. A new specification construct to expand the reach of automated verification of parallel and concurrent programs

4. A discipline for programming in C++ to reduce or eliminate aliases without novel language features and to produce programs that are easy to reason about

1.1 Foundation and Soundness

A provably sound framework for modular reasoning about parallel programs is presented in chapter 3. The framework takes the form of a calculus for effects. The calculus is defined and discussed as a purely algebraic structure before being used as the basis for a programming model in which data abstraction is the norm and fork-join parallel programs have well-defined semantics. The programming model, while nonstandard, is shown to be reasonable and is used to prove that statements that are specified to have non-interfering effects exhibit equivalent behavior to any order of their sequential composition. This result motivates rules for the semantics of parallelism in this programming model and it is used to show the soundness of modular reasoning about fork-join parallel programs in a real programming language in chapters 4 and 5.

1.2 Leveraging Language Restrictions for Data Race Free Parallelism

In chapter 4, a fundamental barrier to automated software verification—the presence of aliased references—is addressed. Aliased references cause problems because it is in general impossible to decide via local reasoning whether two references are aliases to one another. When there is even a possibility of aliasing, determining the scope of effects of, *e.g.*, a method call, becomes intractable. One language that has sidestepped the aliasing problem is RESOLVE, in which most routine aliases are avoided and they are compartmentalized when unavoidable [132]. The alias control mechanism of RESOLVE has been dubbed *clean semantics* because it creates an elegant space for reasoning devoid of complicated frame and heap assertions. Within the framework of RESOLVE, rich data abstractions can be designed to enable the development of obviously data race free parallel programs without the need for additional specification constructs. Doing so amortizes the cost of reasoning about such programs by requiring data race freedom to be shown only once (in the implementation of each data abstraction) and producing client code that requires no proof of non-interference. Several array abstractions are introduced in chapter 4 which permit a client to partition an array into arbitrary sub-arrays based on some client-defined indices and to operate on those sub-arrays in parallel and without fear of interference. Similar data abstractions appear in related work for a variety of programming languages, but their guarantees are weaker than those achieved here, showcasing the power of clean semantics to contribute to the modular verification of parallel programs.

1.3 Abstract Specifications for Non-Interference

To solve the second problem (*i.e.*, how to preserve abstraction and modularity in reasoning about parallel programs), a new specification construct called a non-interference contract is introduced in chapter 5. A non-interference contract can guarantee innterference-free execution of parallel programs that invoke operations on instances of abstract data types. To illustrate the ideas, a bounded queue data abstraction is presented with informal descriptions of three different implementations that vary in their potential for parallelism among different queue operations. To capture the parallel potential in a class of implementations of the queue, its specification is augmented with a non-interference contract that identifies additional details that enable a verifier to make guarantees of safe execution of parallel code. A non-interference contract is still quite abstract and is devoid of concrete implementation details. The novelty of this solution is that it modularizes the verification problem for parallel programs along abstraction boundaries. That is, verification of implementation code with respect to both its behavioral specification and its non-interference specification is done once in the lifetime of the implementation; verification of client code can then rely solely on those specifications.

The rules for reasoning with non-interference contracts generalize results from others [115, 121, 43] by increasing abstraction. Specifically, a non-interference contract can guarantee the interference freedom of several parallel statements even when they share the same abstract variable, as long each of the statements affects a different part of a concrete representation of that variable. The need to delve into implementation details is avoided by a non-interference contract because it enhances the abstract specification to help the verifier of a client program to establish when statements do not interfere.

1.4 Easing Reasoning in a Mainstream Programming Language

The C++11 standard included *move semantics* [78], which changed the way data movement works in certain situations. A primary motivation for adopting move semantics is performance improvement by reducing or eliminating the copying of temporary variables such as return values and expression values. Chapter 6 describes a discipline for programming C++ that leverages move semantics to approximate RESOLVE's clean semantics and, in turn, produce software that is easy to reason about. The motivation for adopting software disciplines in general is well-understood: programming without a strict software discipline produces code that is hard to read, even harder to understand, and nearly impossible to prove correct (if in fact it is correct). Following even the simplest of software disciplines—naming conventions, formatting conventions, *etc.*—dramatically improves the readability and understandability of a program. More sophisticated disciplines, however, are needed for substantial progress towards easing (automated) reasoning. For this reason, many idioms, design patterns, style guides, and language mechanisms have been devised that attempt to improve this facet of software engineering. A common theme among many of these disciplines is *alias control*, in which aliased references are advertised, made immutable, or eliminated altogether. The discipline presented in chapter 6 shares similar objectives, but the approach relies on existing language mechanisms in a language widely used in practice, rather than introducing new capabilities to a language or starting from scratch with an entirely new language.

1.5 Thesis Statement

In summary, this dissertation expands the reach of automatable, modular verification of software to include a class of fork-join parallel programs and provides a sound foundation for continuing that expansion into more complex parallel and concurrent programs. It also provides guidance on how to use a mainstream programming language as a compilation target for verification-focused languages by describing a discipline for programming in C++ that makes extensive use of move semantics. The discipline could also be used as a tool for teaching formal reasoning to beginning computer science students, or for large-scale software development.

Chapter 2: Related Work

There is a large body of work related to this dissertation ranging from closely-related to only tangentially so. The order of examination of those projects is as follows. First, an overview of the earliest and most influential work on formal reasoning about parallel and concurrent programs is presented, then efforts to enable safe parallelism through data abstraction are explored, including array slices and locking data structures. Next, recent and contemporary verification efforts are summarized, including automated verification of sequential programs, auto-active and interactive verification of parallel and sequential programs, and techniques for verifying only that certain properties of parallel and concurrent programs hold. Finally, an overview of programming language-based efforts to ease reasoning and enable safe concurrency is presented.

2.1 Classical Efforts

Owicki and Gries, in their seminal paper [115], provide many of the foundational ideas for verifying the partial correctness of parallel programs. In the paper, they formulate a proof rule for arbitrarily many parallel statements as follows:

$$\frac{\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}, \ldots, \{P_n\}S_n\{Q_n\} \text{ are interference-free}}{\{P_1 \wedge P_2 \wedge \cdots \wedge P_n\}S_1 \| S_2 \| \ldots \| S_n\{Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n\}} \tag{2.1}$$

The definition of "interference-free" used in eq. (2.1) is concerned with the interference-freedom of the *proofs* of the Hoare triples in the premises, and not directly with the statements themselves. This definition is conservative in the sense that it places strict conditions on statements that share modified variables that are weakened by the approach in this dissertation.

Concurrent Separation Logic (CSL) [121, 43], an extension of Separation Logic, is one attempt to address this limitation. In the simplest case, the parallel composition rule in CSL is as follows.

$$\frac{\{P_1\}S_1\{Q_1\} \quad \{P_2\}S_2\{Q_2\} \quad \dots \quad \{P_n\}S_n\{Q_n\}}{\{P_1 * P_2 * \dots * P_n\}S_1 \| S_2 \| \dots \| S_n\{Q_1 * Q_2 * \dots * Q_n\}} \tag{2.2}$$

Here, $*$ is the *separating conjunction*, which asserts that the predicates on either side of the operator hold in disjoint portions of the heap. The semantics of concurrent separation logic are detailed in [43] where restrictions on the statements $S_1, S_2, \dots, S_n$ are given. These restrictions involve access to memory locations from within the statements and amount to a definition of interference freedom that relies on exposing enough implementation details to reason about pointers directly. This exposure limits somewhat the support for modularity: the assertions required to verify programs with even moderately deep abstraction hierarchies are considerably more complex than their shallow-hierarchy counterparts.

One approach supporting *modular* non-interference specifications is rely-guarantee reasoning [86, 105, 87]. In rely-guarantee frameworks, a process *relies* on the environment behavior in order to *guarantee* properties about itself. The required behavior of the environment can be thought of as a generalized form of non-interference. Rely-guarantee methods enable modular reasoning about many kinds of programs including aspect-oriented programs [130], asynchronous programs [70], garbage collectors [146], and non-blocking algorithms [56]. Improving the scalability and expressiveness of rely-guarantee, including combining it with separation logic, is an active area of research [138, 137, 64, 60, 104, 72].

When specification statements as described by Morgan [106] are used to define the semantics for a programming language, the parallel composition rule can be characterized as a lattice join ($\vartriangle$) on these statements [102]. The effect of interference on parallel execution is captured naturally by the definition of $\vartriangle$ in the lattice: that is, the join of interfering statements is "miraculous". Non-interfering statements are those with disjoint modification frames, which limits concurrency to statements on independent variables.

Well-known work by Herlihy and Wing on linearizability [77] laid the foundation for abstract and modular reasoning about parallel programs. Their work established a correctness condition for concurrent objects that guarantees that any concurrent program that operates on a linearizable object will result in a value that may have been achieved by an "equivalent" sequential program on the same object. That an object is linearizable is, however, difficult to prove in general [59, 125]. The results in chapter 3 could be cast as a weakening of the definition of a linearizable object to include only *some* concurrent programs that operate on the object, thus making the condition easier to verify.

2.2 Enabling Parallelism Through Data Abstractions

A central contribution of this research is the recognition that data abstraction and concurrency are both essential software engineering tools and should be addressed jointly. Software component interfaces can be designed such that parallel programs can be written to utilize them and take advantage of lock-free and data-race-free parallelism. It is useful to examine work related to software component design to enable concurrency as a gauge of the utility of the solution presented in chapter 4 as well as to understand the burdens that are introduced or lifted by designing these components in the context of a verification-focused language such as RESOLVE.

2.2.1 Array Slices

Many modern programming languages have libraries that provide data abstractions with operations that divide an array in to several sub-arrays for which it is easy to show comprise disjoint index sets [15, 17, 16, 20, 19, 9]. When a client of the array uses sub-arrays with disjoint index sets as parameters to parallel statements, they have the guarantee that—*absent aliases between elements of the original array*—there will be no data races during concurrent access to distinct slices. However, an important distinction between related work and the work described in chapter 4 is that aliases are guaranteed to be absent by properties of the programming language which serves as the broad context for this dissertation. Without robust alias control, race conditions remain a threat even when the index sets of array slices are disjoint.

2.2.2 Locking Data Structures

Another approach to enabling data abstraction in the face of concurrency is to use data structures that (internally) use locking or synchronization mechanisms to ensure exclusive access by one thread or process at a time. These software components are useful because they enable data-race-free concurrency even when the data abstraction might not, at first glance, seem to allow it. However, programs written with these components are not necessarily deterministic, complicating formal reasoning. Examples of these kinds of components for Java can be found in the `java.util.concurrent` package [114] and for C++ in Mircosoft's PPL [11].

2.3 Verification, Concurrency, and Abstraction

Research on the formal verification of software can be roughly divided into three overlapping areas.

1. Verification of full functional correctness of programs that make use of rich data abstractions. Such research is usually limited to sequential programs because of complications to concurrency introduced by abstraction.

2. Verification of full functional correctness of concurrent programs. Most of this work is dedicated to verifying the correctness of programs that make use of concurrency primitives or other low-level software components, and such proofs are not typically fully automated.

3. Automated verification of a limited set of properties (such as safety, termination, or data race freedom) of concurrent and parallel programs. Efforts in this space include those employed either at compile-time or run-time.

The main contribution of this disertation lies at the intersection of these three areas, that is, it tackles the automated verification of full functional correctness of concurrent programs that make use of data abstractions.

All of the verification efforts discussed here can be placed on a spectrum of interactivity, ranging from fully interactive to fully automatic (with so-called auto-active verification in the middle). The focus of the discussion here is on automatic and auto-active verification; a discussion of the details of the taxonomy of verification efforts can be found in [143].

2.3.1 Verification of Full Functional Correctness of (Sequential) High-Level Programs

Significant advances have been made recently in the verification of high-level programs (including in the context of RESOLVE [126, 139, 132, 143]), some of which are summarized in [76, 95, 90]. While Dafny [100, 101], backed by Z3 [58], uses its own programming language, several other automation efforts are based on industry-standard languages. JML (Java Modeling Language) [96, 97] and related tools such as KeY and ESC/Java2 [48, 34, 35] (for Java), Jahob (for Java) [42, 94, 147, 148], VCC (for C) [53, 107], VeriFast (for C and Java) [85, 8], SPARK (for Ada) [5], Spec# (for C#) [33, 6], and nearly all other recent or active similar projects emphasize the claimed ability to specify and automatically verify programs written in existing languages with extra annotations. Krakatoa [65] (also for JML-annotated Java) and Frama-C's verification plug-in Jessie [3] (for annotated C with ACSL [1]), along with the latter's predecessor (the stand-alone tool Caduceus [65]), compile down to the intermediate language Why [39, 54], from which VCs are derived and ultimately proved using Coq [57].

RESOLVE has been used in long-term and large-scale efforts to automatically verify the full functional correctness of imperative programs [81]. Zaccai, in his PhD thesis, discusses the challenges to automated verification in Java which stem primarily from aliased references (and arguments) [145]. Hollingsworth, *et al*, built relatively large-scale software using a dialect of C++ called RESOLVE/C++ [81]. Their work has validated that many important formal methods principles—such as the ones that drive the development of Clean++ in chapter 6—can be applied at scale in real software. RESOLVE/C++ has also been used to teach software engineering principles to undergraduate students [103, 44, 127, 24].

2.3.2 Automated Verification of Certain Properties of Programs

Several success stories in automated hardware and software verification focus on specific properties (*e.g.*, absence of deadlocks, race conditions, or null pointer accesses). They span sequential and concurrent/distributed software and use symbolic model-checking [46, 47, 52, 55, 57, 61] and other formal techniques. They include several by industry [2, 28, 32, 33, 31, 30, 74, 75, 99, 116, 4, 5, 6]. Static techniques (some cast as extensions of typing) focusing on limiting the inadvertent introduction of races include [63, 109, 23, 69, 67, 108, 88], whereas dynamic techniques which improve precision at the expense of execution overhead include software-based efforts [120, 119, 66, 123, 36, 37, 122] and harware-based efforts such as transactional memory [124, 71].

The reasoning approach introduced in chapter 3 and its subsequent implementation in the form of non-intererence contracts (chapter 5) is related to but dissimilar from the work of region logic, an approach that has been studied to enable shared-memory concurrency in situations where multiple threads might mutate the same variable [29]. One application of region logic is Deterministic Parallel Java (DPJ) [40]. The key feature of interference contracts that distinguishes it from region logic is that in an implementation, it is not memory locations that are assigned to pieces of an object's partition but rather *pieces of the partitions* of those fields. In contrast, a region logic-based specification of the implementation of a software component would identify regions explicitly as disjoint sets of memory locations.

Another capability that differentiates non-interference contracts from DPJ (in particular) is that effects clauses may be conditional on abstract values of parameters. This capability does not identify a *fundamental* novelty beyond (theoretical) region logic, but rather demonstrates the advantages of implementing a specification framework for parallel programs in the context of a language with sequential-program verification capability.

2.4 Specification and Verification with Shared Data

Chapter 6 presents a discipline in which aliases are largely avoided at all costs, primarily through extensive use of `std::unique_ptr`. However, as discussed in that chapter, data sharing—including aliasing—is occasionally unavoidable. The challenge of verifying (*i.e.*, reasoning about) software involving aliasing is well recognized in the literature [98, 121].

A language with explicit references necessarily gives rise to aliases, either through assignment statements or parameter passing. Aliases, however, create significant challenges for reasoning by undermining modularity and by forcing memory addresses to be part of the state of the program [98]. Because of *potential* aliasing, verification in popular languages such as Java requires establishing the frame property (that objects not mentioned do not change) for practically every method call. Some machinery such as separation logic [121, 111] or equivalent is absolutely necessary to ease this process [145]. When there is potential for aliasing, modular reasoning of one component at a time becomes difficult because it is not possible to reason about variables using their abstract values in specifications and instead their data representations become relevant, which breaks abstraction [73]. Though there has been progress, the automation of software verification remains a challenge when code involves objects and aliased references.

Separation logic is an extension of Hoare's logical rules to address the challenges of aliases. Specification of a concurrent map using separation logic is the topic of [144]. Examples of (interactive) verification using separation logic in Coq include [51, 49, 54] and in VeriFast to verify Java and C programs include [84, 85]. Automating verification with separation logic is the topic of [38, 41, 117, 118, 42]. There have been attempts to combine modularization and to address information hiding in conjunction with separation logic [111, 112, 113], though problems remain in generalizing the approach to encompass many object instances.

Another approach to ease reasoning about aliased references is to devise and reuse concepts in which a set of locations (an abstraction of addresses) is "shared" [91, 92, 134]. Two concepts in RESOLVE to address this include a general referencing concept and a specialized acyclic referencing concept, both of which are relatively low-level and provide a reference type and operations for the usual manipulation of references, including linking and aliasing [93]. The detailed specifications employ standard, higher-order logic and their automated verification reuses results from extensions to function theory [133]. The discussion in [91] also contains a solution to avoid the classical parameter aliasing problem on calls with repeated arguments, a solution that can be realized directly in C++ with move semantics.

2.5 Language-Based Approaches

Because the reasoning difficulties aliases introduce are well-known, several popular programming languages have introduced features to keep them under control at the language level. Some efforts have focused on efficient ways to enrich value types so that they may be used for as much software as possible or on developing disciplines and style guidelines within popular, existing languages to control and advertise the use of aliases. Other efforts have been directed toward the development of entirely new languages built from the ground up to eliminate aliasing concerns. Ultimately, not all aliasing can be avoided, so admitting aliasing and enabling sound reasoning in its presence are topics that have received attention in the formal methods literature and in the programming languages community.

2.5.1 Efficient Value Types

One typical solution to the aliasing problem offered by modern programming languages is the introduction of value types [26, 18, 22] that, when assigned, copy the variable's entire value to the destination.[1] The obvious advantage of value types in the context of this work is that aliases are never introduced, thus enabling simpler reasoning. Unfortunately, frequent copying is quite costly for large objects. This has

[1]At least, this is the client's view of what is happening. Some types, such as immutable ones, need not be copied entirely.

led some languages to impose restrictions on what may be done with variables of value types so their use does not incur unaccaptable performance penalties [26, 14].

For example, Swift [26] offers a rich toolbox for creating and using value types (called `structs`). However, value types in Swift are immutable by default and the compiler restricts them from being mutated without additional annotations. This limitation is necessary because structs in Swift are copied lazily (a mechanism sometimes called "copy-on-write"), so making them immutable allows Swift to offer equivalent performance in struct-heavy code as it does in reference-heavy code (except in rare cases where a struct is mutated). Implementing lazy copying at the language level encourages programmers to consider using value types where appropriate without having to worry about performance. Other work on lazy copying includes work by Adcock on implementing it while maintaining value semantics in RESOLVE/C++ [25].

2.5.2 The Google C++ Style Guide

Google's style guide [12] offers extensive guidance on how to write disciplined C++ code, from variable naming conventions to high-level structuring. Several sections address themes directly related to the work in chapter 6.

Of most relevance is the section "Ownership and Smart Pointers". In that section, an argument is made as to why having a regimented discipline for expressing ownership (beyond simple pointers) is a "Good Thing" in C++ programs. The stylistic suggestion made in the guide is to "prefer to keep ownership with the code that allocated it," and when ownership transfer is necessary, to use `std::unique_ptr` to make such transfer explicit.

Two other sections, “Copyable and Movable Types” and “Rvalue References” are also closely related to the discipline described in chapter 6. Google’s C++ style guide recommends that a class be either copyable or movable or neither, but not both copyable and movable except in rare circumstances. The justification for this rule is that having a copyable class that is also movable is a potential source of bugs because the expected (*i.e.*, copying) behavior of parameter values or return values might not actually occur. The guide argues that defining moving operations on a copyable class should be viewed strictly as a performance optimization and so should not be used unless there is a clear performance improvement for doing so. The section on rvalue references makes the recommendation that rvalue references (identified in C++ by `&&`) be used in overloaded function pairs (one taking a **`const`**`&` and the other a `&&` argument). Again, the crux of the decision is that rvalue references and move semantics should be viewed strictly as a performance optimization. Minimizing aliasing and simplifying reasoning are not considered in the discussion.

2.5.3 The Rust Programming Language

Rust [89], an open-source programming language project sponsored by Mozilla, is a relatively new and popular programming language that aims to (and does, in many cases) solve exactly the kind of problems identified in chapter 6. However, the motivations for it are related to memory safety, not based in a desire for simple (abstract) reasoning.

Specifically, the developers of Rust identified that shared data (*e.g.*, in the form of aliases) is a primary source of problems such as dangling pointers and memory leaks. It aims to solve these problems by introducing a statically-checked system of ownership and borrowing to ensure that immutable data is never mutated except when the programmer really, *really* wants that behavior. Rust's system of ownership and borrowing have deep implications for alias control in programs and is quite nuanced, so it will be discussed only at a high level here. Ownership in Rust follows three rules:

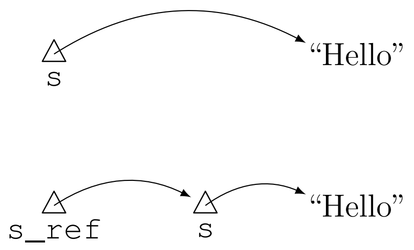
1. Each value in Rust has a variable called its *owner*
2. A value may have only one owner at a time
3. When an owner goes out of scope, its value is dropped (*i.e.*, its memory is freed)

Assignment in Rust, by default, transfers ownership from the right hand side to the left hand side. The semantics of Rust's ownership transfer is similar—though not identical—to the semantics of assignment with the `std::move` operation in C++. Both Rust and C++ leave a moved-from right hand side with an undefined value; the difference is that in Rust, a compile-time check ensures that no moved-from variable is subsequently used before it is given a new value. A C++ compiler makes no such checks.

In Rust, a variable may *borrow* ownership from another, in which case the second variable is a reference to the first (borrowing does not introduce an alias directly—see fig. 2.1). Borrows may be done mutably or immutably, and the rules for each vary. Overall, borrowing in Rust follows three core rules:

1. There may be either one mutable reference or any number of immutable references in scope for a given variable
2. A reference must always be valid (the value it refers to must not have been dropped)

3. Assignment may only be done to variables that are not borrowed from



s owns the value "Hello".

s_ref is borrowing the value "Hello" from s.

Figure 2.1: Visualizing borrowing in Rust.

In Rust, shared data is advertised (through the borrowing operator `&`) when needed so that the programmer can reason more carefully about those portions of code. All of the ownership and borrowing rules in Rust are enforced at compile time.

2.5.4 Pointer-Free Parallel Programming in ParaSail

ParaSail [135] addresses several of the issues raised in this dissertation, but in the context of a new language. ParaSail was developed with two complementary goals in mind: pointer-free and parallel programming. ParaSail unifies pointer-free and parallel programming idioms by permitting the sharing of data across threads explicitly only in cases where it is intended. Everywhere else there are no aliases.

2.5.5 Functional Languages and Immutable Data

A key insight of functional languages is that the exclusive use of immutable data can eliminate a whole class of reasoning problems. Immutable types, of course, are not limited to functional languages: as mentioned above, structs in Swift are immutable by default. It is functional languages' *exclusive* use of immutable types that makes them so attractive as targets for verification efforts [131]. While functional programming has its benefits, the focus of this work is on developing and verifying imperative programs.

Chapter 3: Foundation and Soundness of Abstract Non-Interference Specifications and Proof Rules for Parallel Operation Calls

Parallel programs are notoriously difficult to prove correct because of the possibility of interference among threads. The formal verification of such programs typically involves reasoning directly about the implementation of the objects in use rather than relying solely on their behavioral specification. However, complex software components in the real world are built in a layered fashion, reusing complex high-level components (which themselves are built by reusing other high-level components). Verification of the correctness of sequential programs built in this manner typically proceeds in a modular fashion, using the behavioral specification—and not the implementation—of the components that are used. Unfortunately, such modular reasoning breaks down in a concurrent system because the underlying implementation matters somewhat. For example, a reasonable behavioral specification for a queue could be implemented in many ways, one of which might become corrupted when multiple processes attempt to concurrently enqueue and another that guarantees a consistent state no matter how many concurrent processes are modifying it. Of course, only the second implementation would be useful in a concurrent system.

This chapter proposes a sound basis for enhancing a behvarioral specification with a non-interference specification to capture the conditions under which it is safe to use a component in a parallel context. The verification that an implementation meets such a non-interference specification can proceed modularly and without sacrificing abstraction (this advances the state of the art in the verification of parallel programs). It is shown that the manner in which an implementation meets its non-interference specification and is considered "safe" implies that the parallel execution of non-interfering statements is equivalent to all possible sequential executions of those statements.

3.1 A/P Calculus

A non-interference specification must identify, for each operation on an object, under what conditions various pieces of the object's state are *affected* (*i.e.*, might have their value changed), *preserved* (*i.e.*, might have their value read), or *ignored* (*i.e.*, are neither read from nor written to) by the execution of that operation. This section introduces a calculus to facilitate formal reasoning about these effects independent of their use in programs. An effect is modeled as a pair of sets (A, P); despite the apparent generality in the calculus, effects find their primary use in reasoning about programs involving partitioned objects, in which some pieces of each object are affected (corresponding to the 'A' set) and others are preserved (the 'P' set). Pieces of an object that are ignored by an effect are not in either the A or P sets of that effect.

For example, consider the C++ program fragment in listing 3.1. The method `modifyX` changes the value of `p.x` and reads but does not change the values of `p.y`

Listing 3.1: C++ program fragment to illustrate a simple use of effects.

```
struct Point {
  int x;
  int y;
  int z;
};

void modifyX(struct Point &p) {
  p.x = p.x + p.y + p.z;
}
```

and `p.z`. In the A/P Calculus, its effect is summarized as the pair $(\{p.x\}, \{p.y, p.z\})$. In this way, some information about the method's implementation is exposed to the client: just enough information that the client can make a decision about how to safely use the method in a parallel program. Of course, objects in real-world programs are typically far more complex than the `Point` struct in listing 3.1. The A/P Calculus is general enough to serve as a useful reasoning tool both for simple structures such as `Point` and for complex objects with rich abstraction of the kind used in real-world programs. This generality is applied through the mechanisms described in sections 3.2 and 3.3.

This section is organized as follows. First, several definitions and simple lemmas provide the structure of effects. That structure is then shown to be a lattice, and finally a variety of interesting results about effects are proved.

3.1.1 Basic Definitions

Definition 3.1.1 (Effect)**.** An *effect* is a pair of sets $e = (A, P)$.

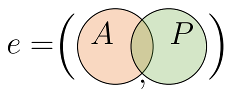


Figure 3.1: Visualization of the definition of an Effect.

Definition 3.1.2 (Target of an effect)**.** The *target* of an effect $e = (A, P)$ is $\mathcal{T}(e) = A \cup P$.

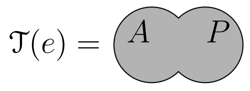


Figure 3.2: Visualization of the Target function.

Definition 3.1.3 (Reduction)**.** The *reduction* of an effect $e = (A, P)$, denoted $\mathcal{R}(e)$, is defined as follows.

$$\mathcal{R}(e) = (A, P \setminus A)$$

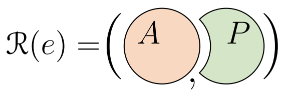


Figure 3.3: Visualization of the Reduction function.

Definition 3.1.4 (Equivalence Modulo Reduction)**.** The relation $\equiv_{\mathcal{R}}$, defined as follows, is an equivalence relation:

$$e_1 \equiv_{\mathcal{R}} e_2 \Leftrightarrow \mathcal{R}(e_1) = \mathcal{R}(e_2)$$

Definition 3.1.5 (Projection)**.** The *projection* of an effect $e = (A, P)$ to a set S is $e|_S = (A \cap S, P \cap S)$.

Definition 3.1.6 (Combined effect)**.** The *combined effect* of two effects $e_1 = (A_1, P_1)$ and $e_2 = (A_2, P_2)$, denoted $e_1 \sqcup e_2$, is defined as follows.

$$e_1 \sqcup e_2 = \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big)$$

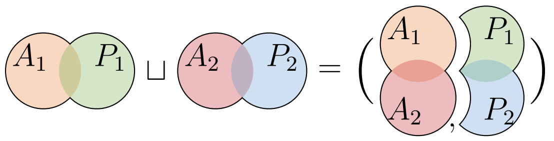


Figure 3.4: Visualization of the Combined effect operation.

Definition 3.1.7 (Common effect)**.** The *common effect* of two effects $e_1 = (A_1, P_1)$ and $e_2 = (A_2, P_2)$, denoted $e_1 \sqcap e_2$, is defined as follows.

$$e_1 \sqcap e_2 = \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big)$$

Definition 3.1.8 (The Is Covered By relation)**.** The *is covered by* relation $\sqsubseteq$ on effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$ is defined as follows.

$$e_1 \sqsubseteq e_2 \Leftrightarrow (A_1 \subseteq A_2) \wedge \big(\mathcal{T}(e_1) \subseteq \mathcal{T}(e_2)\big)$$

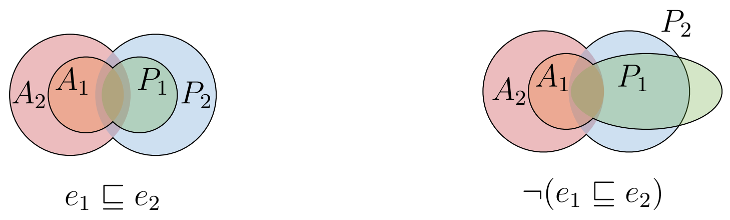


Figure 3.5: Visualization of the Is Covered By relation.

Definition 3.1.9 (The Does Not Interfere With relation)**.** The *does not interfere with* relation $\ddagger$ on effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$ is defined as follows.

$$e_1 \ddagger e_2 \Leftrightarrow \big(A_1 \cap \mathcal{T}(e_2) = \emptyset\big) \wedge \big(A_2 \cap \mathcal{T}(e_1) = \emptyset\big)$$

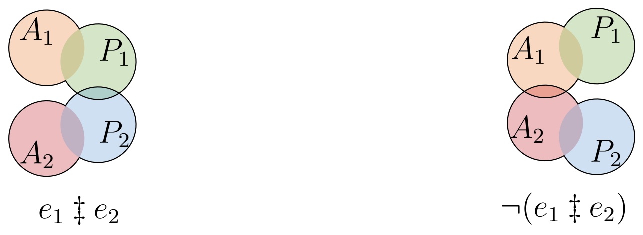


Figure 3.6: Visualization of the Does Not interfere With relation.

Definition 3.1.10 (Well-formedness)**.** An effect $e = (A, P)$ is *well-formed* if $A \cap P = \emptyset$. e is *well-formed with respect to* a set S if e is well-formed and $\mathcal{T}(e) \subseteq S$.

3.1.2 Basic Lemmas

Lemma 3.1.1. *For effect $e = (A, P)$, $\mathcal{T}\big(\mathcal{R}(e)\big) = \mathcal{T}(e)$.*

Proof.

$$\begin{aligned}
\mathcal{T}\big(\mathcal{R}(e)\big) &= \mathcal{T}\big((A, P \setminus A)\big) && \text{(def. of } \mathcal{R}\text{)} \\
&= A \cup (P \setminus A) && \text{(def. of } \mathcal{T}\text{)} \\
&= A \cup P && \text{(appl. of } \cup, \setminus\text{)} \\
&= \mathcal{T}(e) && \text{(def. of } \mathcal{T}\text{)}
\end{aligned}$$

□

Lemma 3.1.2. *For effects* $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$, $\big(e_1 \sqcup \mathcal{R}(e_2) = e_1 \sqcup e_2\big)$ *and* $\big(e_1 \sqcap \mathcal{R}(e_2) = e_1 \sqcap e_2\big)$. *That is, when computing* $\sqcup$ *or* $\sqcap$, $\mathcal{R}$ *need only be applied once, at the end, rather than at each individual step.*

Proof.

$$\begin{aligned}
e_1 \sqcup \mathcal{R}(e_2) &= \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(\mathcal{R}(e_2)))\big) && \text{(def. of } \sqcup\text{)} \\
&= \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(lemma 3.1.1)} \\
&= e_1 \sqcup e_2 && \text{(def. of } \sqcup\text{)}
\end{aligned}$$

$$\begin{aligned}
e_1 \sqcap \mathcal{R}(e_2) &= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(\mathcal{R}(e_2)))\big) && \text{(def. of } \sqcap\text{)} \\
&= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) && \text{(lemma 3.1.1)} \\
&= e_1 \sqcap e_2 && \text{(def. of } \sqcap\text{)}
\end{aligned}$$

□

Lemma 3.1.3. *For effect* $e = (A, P)$, $\mathcal{R}(e)$ *is well-formed. That is,* $\mathcal{R}$ *is a transformation from ill-formed effects to well-formed effects.*

Proof.

$$
\begin{aligned}
&A \cap (P \setminus A) = \emptyset && \text{(appl. of } \setminus, \cap) \\
&\quad \Rightarrow (A, P \setminus A) \text{ is well-formed} && \text{(def. of well-formedness)} \\
&\quad \Rightarrow \mathcal{R}(e) \text{ is well-formed} && \text{(def. of } \mathcal{R})
\end{aligned}
$$

□

Lemma 3.1.4. *Well-formedness is closed under* $|$, $\mathcal{R}$, $\sqcup$, *and* $\sqcap$.

Proof. We show each individually.

1. Well-formedness is closed under $|$. That is, for any well-formed effect $e = (A, P)$ and set S, $e|_S$ is well-formed.

$$
\begin{aligned}
&e|_S = (A \cap S, P \cap S) && \text{(def. of } |) \\
&\quad \Rightarrow (A \cap S) \cap (P \cap S) = (A \cap P) \cap (S \cap S) && \text{(assoc., commut. of } \cap) \\
&\quad \Rightarrow (A \cap S) \cap (P \cap S) = \emptyset \cap (S \cap S) && \text{(well-formedness of } e) \\
&\quad \Rightarrow (A \cap S) \cap (P \cap S) = \emptyset && \text{(appl. of } \cap) \\
&\quad \Rightarrow (A \cap S, P \cap S) \text{ is well-formed} && \text{(def. of well-formedness)} \\
&\quad \Rightarrow e|_S \text{ is well-formed} && \text{(def. of } |)
\end{aligned}
$$

2. Well-formedness is closed under $\mathcal{R}$. That is, for any well-formed effect e, $\mathcal{R}(e)$ is well-formed.

   By lemma 3.1.3, the reduction of any effect (including well-formed ones) is well-formed. Therefore, well-formedness is closed under $\mathcal{R}$.

3. Well-formedness is closed under $\sqcup$ and $\sqcap$. That is, for any well-formed effects e_1, e_2, $e_1 \sqcup e_1$ and $e_1 \sqcap e_2$ are both well-formed.

By the definitions of $\sqcup$ and $\sqcap$, $e_1 \sqcup e_2$ and $e_1 \sqcap e_2$ are each the result of applying $\mathcal{R}$ to some effect. By lemma 3.1.3, the reduction of any effect is well-formed. Therefore well-formedness is closed under $\sqcup$ and $\sqcap$.

□

Lemma 3.1.5. *The operations $\sqcup$ and $\sqcap$ are associative and commutative.*

Proof. 1. $\sqcup$ is associative. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$:

$$\begin{aligned}
& e_1 \sqcup (e_2 \sqcup e_3) \\
&\quad = e_1 \sqcup \mathcal{R}\big((A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3))\big) && \text{(def. of } \sqcup\text{)} \\
&\quad = e_1 \sqcup \big(A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3)\big) && \text{(lemma 3.1.2)} \\
&\quad = \mathcal{R}\big((A_1 \cup (A_2 \cup A_3), \mathcal{T}(e_1) \cup (\mathcal{T}(e_2) \cup \mathcal{T}(e_3)))\big) && \text{(def. of } \sqcup\text{)} \\
&\quad = \mathcal{R}\big(((A_1 \cup A_2) \cup A_3, (\mathcal{T}(e_1) \cup \mathcal{T}(e_2)) \cup \mathcal{T}(e_3))\big) && \text{(assoc. of } \cup\text{)} \\
&\quad = \big(A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2)\big) \sqcup e_3 && \text{(def. of } \sqcup\text{)} \\
&\quad = \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) \sqcup e_3 && \text{(lemma 3.1.2)} \\
&\quad = (e_1 \sqcup e_2) \sqcup e_3 && \text{(def. of } \sqcup\text{)}
\end{aligned}$$

2. $\sqcup$ is commutative. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$:

$$\begin{aligned}
e_1 \sqcup e_2 &= \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(def. of } \sqcup\text{)} \\
&= \mathcal{R}\big((A_2 \cup A_1, \mathcal{T}(e_2) \cup \mathcal{T}(e_1))\big) && \text{(commut. of } \cup\text{)} \\
&= e_2 \sqcup e_1 && \text{(def. of } \sqcup\text{)}
\end{aligned}$$

3. $\sqcap$ is associative. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$:

$$\begin{aligned}
&e_1 \sqcap (e_2 \sqcap e_3) \\
&\quad = e_1 \sqcap \mathcal{R}\big((A_2 \cap A_3, \mathcal{T}(e_2) \cap \mathcal{T}(e_3))\big) && \text{(def. of } \sqcap\text{)} \\
&\quad = e_1 \sqcap \big(A_2 \cap A_3, \mathcal{T}(e_2) \cap \mathcal{T}(e_3)\big) && \text{(lemma 3.1.2)} \\
&\quad = \mathcal{R}\big((A_1 \cap (A_2 \cap A_3), \mathcal{T}(e_1) \cap (\mathcal{T}(e_2) \cap \mathcal{T}(e_3)))\big) && \text{(def. of } \sqcap\text{)} \\
&\quad = \mathcal{R}\big(((A_1 \cap A_2) \cap A_3, (\mathcal{T}(e_1) \cap \mathcal{T}(e_2)) \cap \mathcal{T}(e_3))\big) && \text{(assoc. of } \cap\text{)} \\
&\quad = \big(A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2)\big) \sqcap e_3 && \text{(def. of } \sqcap\text{)} \\
&\quad = \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) \sqcap e_3 && \text{(lemma 3.1.2)} \\
&\quad = (e_1 \sqcap e_2) \sqcap e_3 && \text{(def. of } \sqcap\text{)}
\end{aligned}$$

4. $\sqcap$ is commutative. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$:

$$\begin{aligned}
e_1 \sqcap e_2 &= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) && \text{(def. of } \sqcap\text{)} \\
&= \mathcal{R}\big((A_2 \cap A_1, \mathcal{T}(e_2) \cap \mathcal{T}(e_1))\big) && \text{(commut. of } \cap\text{)} \\
&= e_2 \sqcap e_1 && \text{(def. of } \sqcap\text{)}
\end{aligned}$$

□

3.1.3 The A/P Lattice

Effects form a lattice with join and meet operations $\sqcup$ and $\sqcap$. While the lattice is defined here only for well-formed effects, it could be analogously defined for equivalence classes of effects under $\equiv_{\mathcal{R}}$, and indeed that formulation might be preferred for some applications.

Notation

Beginning in this section we adopt the following definitions and notational conveniences, which are used for the remainder of this chapter.

- S is some set
- $E_S = \{e : e \text{ is a well-formed effect with respect to } S\}$ (when S is understood or irrelevant, it is left out and we refer simply to E)
- $\bot = (\emptyset, \emptyset)$ is the empty effect
- $\top_S = (S, \emptyset)$ is the total effect (when S is understood or irrelevant, it is left out and we refer simply to $\top$)
- $\mathfrak{L}_S = (E_S, \sqcup, \sqcap, \bot, \top_S)$ (when S is understood or irrelevant, it is left out and we refer simply to $\mathfrak{L}$)

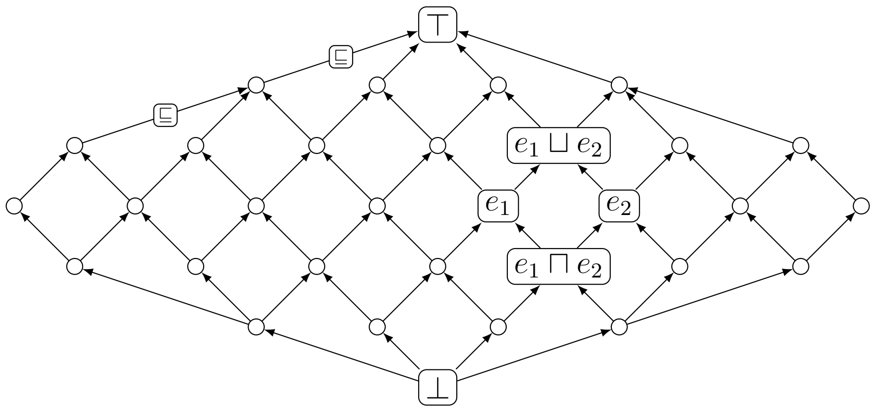


Figure 3.7: The effects lattice $\mathfrak{L}$.

Theorem 1 (Well-formed effects form a lattice)**.** *For all S, $\mathfrak{L}_S$ is a bounded lattice over E_S with operations $\sqcup$ and $\sqcap$, least element $\bot$, and greatest element $\top_S$.*

Proof. We show that $(E, \sqcup, \sqcap)$ is a lattice, then show that $\bot$ and $\top$ are its bounds.

First, by lemma 3.1.4, E is closed under $\sqcup$ and $\sqcap$. (To be precise, the lemma states that the set of *all* well-formed effects is closed under those operations. However, the proofs that $\mathcal{T}(e_1 \sqcup e_2) \subseteq S$ and that $\mathcal{T}(e_1 \sqcap e_2) \subseteq S$ for $e_1, e_2 \in E_S$ are trivial.)

Claim 1. $(E, \sqcup, \sqcap)$ *is a lattice.*

Proof. By lemma 3.1.5, the operations $\sqcup$ and $\sqcap$ are associative and commutative. For these operations to form a lattice, they must additionally satisfy the absorption properties for effects $e_1 = (A_1, P_1)$ and $e_2 = (A_2, P_2)$ in E:

1. $e_1 \sqcup (e_1 \sqcap e_2) = e_1$

$$\begin{aligned}
& e_1 \sqcup (e_1 \sqcap e_2) \\
&\quad = e_1 \sqcup \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) && \text{(def. of } \sqcap\text{)} \\
&\quad = e_1 \sqcup \big(A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2)\big) && \text{(lemma 3.1.2)} \\
&\quad = \mathcal{R}\big((A_1 \cup (A_1 \cap A_2), \mathcal{T}(e_1) \cup (\mathcal{T}(e_1) \cap \mathcal{T}(e_2)))\big) && \text{(def. of } \sqcup\text{)} \\
&\quad = \mathcal{R}\big((A_1, \mathcal{T}(e_1))\big) && \text{(appl. of } \cap, \cup\text{)} \\
&\quad = \mathcal{R}\big((A_1, A_1 \cup P_1)\big) && \text{(def. of } \mathcal{T}\text{)} \\
&\quad = (A_1, (A_1 \cup P_1) \setminus A_1) && \text{(def. of } \mathcal{R}\text{)} \\
&\quad = \big(A_1, (A_1 \setminus A_1) \cup (P_1 \setminus A_1)\big) && \text{(distrib. of } \setminus\text{)} \\
&\quad = (A_1, P_1 \setminus A_1) && \text{(appl. of } \setminus, \cup\text{)} \\
&\quad = (A_1, P_1) && \text{(def. of well-formedness, appl. of } \setminus\text{)} \\
&\quad = e_1 && \text{(def. of effect)}
\end{aligned}$$

2. $e_1 \sqcap (e_1 \sqcup e_2) = e_1$

$$\begin{aligned}
&e_1 \sqcap (e_1 \sqcup e_2) \\
&\quad = e_1 \sqcap \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(def. of } \sqcup) \\
&\quad = e_1 \sqcap \big(A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2)\big) && \text{(lemma 3.1.2)} \\
&\quad = \mathcal{R}\big((A_1 \cap (A_1 \cup A_2), \mathcal{T}(e_1) \cap (\mathcal{T}(e_1) \cup \mathcal{T}(e_2)))\big) && \text{(def. of } \sqcap) \\
&\quad = \mathcal{R}\big((A_1, \mathcal{T}(e_1))\big) && \text{(appl. of } \cap, \cup) \\
&\quad = \mathcal{R}\big((A_1, A_1 \cup P_1)\big) && \text{(def. of } \mathcal{T}) \\
&\quad = (A_1, (A_1 \cup P_1) \setminus A_1) && \text{(def. of } \mathcal{R}) \\
&\quad = \big(A_1, (A_1 \setminus A_1) \cup (P_1 \setminus A_1)\big) && \text{(distrib. of } \setminus) \\
&\quad = (A_1, P_1 \setminus A_1) && \text{(appl. of } \setminus, \cup) \\
&\quad = (A_1, P_1) && \text{(def. of well-formedness, appl. of } \setminus) \\
&\quad = e_1 && \text{(def. of effect)}
\end{aligned}$$

Therefore, $(E, \sqcup, \sqcap)$ is a lattice. ■

Claim 2. *The empty effect* $\bot = (\emptyset, \emptyset)$ *is the identity of* $\sqcup$*. That is,* $\forall e \in E : e \sqcup \bot = e$*.*

Proof. $\bot \in E$ because $\emptyset \cap \emptyset = \emptyset$ (so $\bot$ is well-formed) and $\emptyset \cup \emptyset \subseteq S$ (so $\mathcal{T}(\bot) \subseteq S$). Given effect $e = (A, P)$ s.t. $e \in E$,

$$\begin{aligned}
e \sqcup \bot &= \mathcal{R}\big((A \cup \emptyset, \mathcal{T}(e) \cup \mathcal{T}(\bot))\big) && \text{(def. of } \sqcup) \\
&= \mathcal{R}\big((A \cup \emptyset, \mathcal{T}(e) \cup \emptyset)\big) && \text{(def. of } \mathcal{T}) \\
&= \mathcal{R}\big((A, \mathcal{T}(e))\big) && \text{(identity of } \cup) \\
&= (A, \mathcal{T}(e) \setminus A) && \text{(def. of } \mathcal{R}) \\
&= (A, (A \cup P) \setminus A) && \text{(def. of } \mathcal{T})
\end{aligned}$$

$$
\begin{aligned}
&= \big(A, (A \setminus A) \cup (P \setminus A)\big) && \text{(distrib. of } \setminus\text{)} \\
&= \big(A, \emptyset \cup (P \setminus A)\big) && \text{(appl. of } \setminus\text{)} \\
&= (A, P \setminus A) && \text{(identity of } \cup\text{)} \\
&= (A, P) && \text{(def. of well-formedness, appl. of } \setminus\text{)} \\
&= e && \text{(def. of effect)}
\end{aligned}
$$

■

Claim 3. *The total effect $\top = (S, \emptyset)$ is the identity of $\sqcap$. That is, $\forall e \in E : e \sqcap \top = e$.*

Proof. $\top \in E$ because $S \cap \emptyset = \emptyset$ (so $\top$ is well-formed) and $S \cup \emptyset \subseteq S$ (so $\mathfrak{T}(\top) \subseteq S$). Given effect $e = (A, P)$ s.t. $e \in E$,

$$
\begin{aligned}
e \sqcap \top &= \mathfrak{R}\big((A \cap S, \mathfrak{T}(e) \cap \mathfrak{T}(\top))\big) && \text{(def. of } \sqcap\text{)} \\
&= \mathfrak{R}\big((A \cap S, \mathfrak{T}(e) \cap S)\big) && \text{(def. of } \top\text{)} \\
&= \mathfrak{R}\big((A, \mathfrak{T}(e))\big) && \text{(appl. of } \cap\text{)} \\
&= (A, \mathfrak{T}(e) \setminus A) && \text{(def. of } \mathfrak{R}\text{)} \\
&= (A, (A \cup P) \setminus A) && \text{(def. of } \mathfrak{T}\text{)} \\
&= \big(A, (A \setminus A) \cup (P \setminus A)\big) && \text{(distrib. of } \setminus\text{)} \\
&= \big(A, \emptyset \cup (P \setminus A)\big) && \text{(appl. of } \setminus\text{)} \\
&= (A, P \setminus A) && \text{(identity of } \cup\text{)} \\
&= (A, P) && \text{(def. of well-formedness, appl. of } \setminus\text{)} \\
&= e && \text{(def. of effect)}
\end{aligned}
$$

■

Therefore, $\mathfrak{L}_S = (E_S, \sqcup, \sqcap, \bot, \top_S)$ is a bounded lattice. □

Theorem 2 (Ordering on $\mathfrak{L}$)**.** *For all S, $\sqsubseteq$ is a partial order induced by $\mathfrak{L}_S$.*

Proof. To show that $\sqsubseteq$ orders $\mathfrak{L}_S$ for any S, it suffices to show that

$$\forall e_1, e_2 : e_1, e_2 \in E_S : e_1 \sqsubseteq e_2 \Leftrightarrow e_1 \sqcup e_2 = e_2.$$

$\Rightarrow$

$$\begin{aligned}
&e_1 \sqsubseteq e_2 \Rightarrow \\
&\qquad e_1 \sqcup e_2 = \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(def. of } \sqcup\text{)} \\
&\qquad = \mathcal{R}\big((A_2, \mathcal{T}(e_2))\big) && \text{(def. of } \sqsubseteq\text{, appl. of } \cup\text{)} \\
&\qquad = \mathcal{R}\big((A_2, A_2 \cup P_2)\big) && \text{(def. of } \mathcal{T}\text{)} \\
&\qquad = (A_2, (A_2 \cup P_2) \setminus A_2) && \text{(def. of } \mathcal{R}\text{)} \\
&\qquad = \big(A_2, (A_2 \setminus A_2) \cup (P_2 \setminus A_2)\big) && \text{(distrib. of } \setminus\text{)} \\
&\qquad = \big(A_2, \emptyset \cup (P_2 \setminus A_2)\big) && \text{(appl. of } \setminus\text{)} \\
&\qquad = (A_2, P_2 \setminus A_2) && \text{(identity of } \cup\text{)} \\
&\qquad = (A_2, P_2) && \text{(def. of well-formedness, appl. of } \setminus\text{)} \\
&\qquad = e_2 && \text{(def. of effect)}
\end{aligned}$$

$\Leftarrow$

$$\begin{aligned}
&e_1 \sqcup e_2 = e_2 \\
&\quad \Rightarrow \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) = e_2 && \text{(def. of } \sqcup\text{, effect)} \\
&\quad \Rightarrow \big(A_1 \cup A_2, (\mathcal{T}(e_1) \cup \mathcal{T}(e_2)) \setminus (A_1 \cup A_2)\big) = e_2 && \text{(lemma 3.1.1)} \\
&\quad \Rightarrow A_1 \cup A_2 = A_2 \wedge \big(\mathcal{T}(e_1) \cup \mathcal{T}(e_2)\big) \setminus (A_1 \cup A_2) = P_2 && \text{(def. of effect)} \\
&\quad \Rightarrow A_1 \cup A_2 = A_2 \wedge \big((A_1 \cup P_1) \cup (A_2 \cup P_2)\big) \setminus (A_1 \cup A_2) = P_2 && \text{(def. of } \mathcal{T}\text{)}
\end{aligned}$$

$$\Rightarrow A_1 \cup A_2 = A_2 \wedge \big((A_1 \cup A_2) \cup (P_1 \cup P_2)\big) \setminus (A_1 \cup A_2) = P_2 \quad \text{(assoc., commut. of } \cup)$$

$$\Rightarrow A_1 \cup A_2 = A_2 \wedge \big(A_2 \cup (P_1 \cup P_2)\big) \setminus A_2 = P_2 \quad \text{(substitution)}$$

$$\Rightarrow A_1 \subseteq A_2 \wedge \big(A_2 \cup (P_1 \cup P_2)\big) \setminus A_2 = P_2 \quad \text{(appl. of } \cup)$$

$$\Rightarrow A_1 \subseteq A_2 \wedge P_1 \cup P_2 = P_2 \quad \text{(appl. of } \setminus)$$

$$\Rightarrow A_1 \subseteq A_2 \wedge P_1 \subseteq P_2 \quad \text{(appl. of } \cup)$$

$$\Rightarrow A_1 \subseteq A_2 \wedge (A_1 \cup P_1) \subseteq (A_2 \cup P_2) \quad \text{(appl. of } \cup)$$

$$\Rightarrow A_1 \subseteq A_2 \wedge \mathcal{T}(e_1) \subseteq \mathcal{T}(e_2) \quad \text{(def. of } \mathcal{T})$$

$$\Rightarrow e_1 \sqsubseteq e_2 \quad \text{(def. of } \sqsubseteq)$$

□

Theorem 3 ($\mathfrak{L}$ is distributive)**.** *For all well-formed effects* $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$*:*

$$e_1 \sqcap (e_2 \sqcup e_3) = (e_1 \sqcap e_2) \sqcup (e_1 \sqcap e_3)$$

Proof.

$$e_1 \sqcap (e_2 \sqcup e_3) = e_1 \sqcap \mathcal{R}\big((A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3))\big) \quad \text{(def. of } \sqcup)$$

$$= e_1 \sqcap \big(A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3)\big) \quad \text{(lemma 3.1.2)}$$

$$= \mathcal{R}\big((A_1 \cap (A_2 \cup A_3), \mathcal{T}(e_1) \cap (\mathcal{T}(e_2) \cup \mathcal{T}(e_3)))\big) \quad \text{(def. of } \sqcap)$$

$$= \mathcal{R}\big(((A_1 \cap A_2) \cup (A_1 \cap A_3), (\mathcal{T}(e_1) \cap \mathcal{T}(e_2)) \cup (\mathcal{T}(e_1) \cap \mathcal{T}(e_3)))\big) \quad \text{(dist. of } \cap)$$

$$= \big(A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2)\big) \sqcup \big(A_1 \cap A_3, \mathcal{T}(e_1) \cap \mathcal{T}(e_3)\big) \quad \text{(def. of } \sqcup)$$

$$= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) \sqcup \mathcal{R}\big((A_1 \cap A_3, \mathcal{T}(e_1) \cap \mathcal{T}(e_3))\big) \quad \text{(lemma 3.1.2)}$$

$$= (e_1 \sqcap e_2) \sqcup (e_1 \sqcap e_3) \quad \text{(def. of } \sqcap)$$

□

3.1.4 Other Lemmas about Effects

Lemma 3.1.6. *For effect e, $\mathcal{R}(e) \sqsubseteq e$.*

Proof. Let $\mathcal{R}(e) = (A_{\mathcal{R}}, P_{\mathcal{R}})$ and $e = (A, P)$.

$$A_{\mathcal{R}} = A \wedge \mathcal{T}\big(\mathcal{R}(e)\big) = \mathcal{T}(e) \qquad \text{(def. of } \mathcal{R}\text{, lemma 3.1.1)}$$

$$\Rightarrow A_{\mathcal{R}} \subseteq A \wedge \mathcal{T}\big(\mathcal{R}(e)\big) \subseteq \mathcal{T}(e) \qquad \text{(appl. of } \subseteq\text{)}$$

$$\Rightarrow \mathcal{R}(e) \sqsubseteq e \qquad \text{(def. of } \sqsubseteq\text{)}$$

□

Lemma 3.1.7. *The ‡ relation is symmetric. That is, for effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$, $e_1 \ddagger e_2 \Leftrightarrow e_2 \ddagger e_1$.*

Proof.

$$e_1 \ddagger e_2$$

$$\Leftrightarrow \big(A_1 \cap \mathcal{T}(e_2) = \emptyset\big) \wedge \big(A_2 \cap \mathcal{T}(e_1) = \emptyset\big) \qquad \text{(def. of } \ddagger\text{)}$$

$$\Leftrightarrow \big(A_2 \cap \mathcal{T}(e_1) = \emptyset\big) \wedge \big(A_1 \cap \mathcal{T}(e_2) = \emptyset\big) \qquad \text{(commut. of } \wedge\text{)}$$

$$\Leftrightarrow e_2 \ddagger e_1 \qquad \text{(def. of } \ddagger\text{)}$$

□

Lemma 3.1.8. *For effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$,*

$$e_1 \ddagger e_2 \wedge e_3 \sqsubseteq e_1 \Rightarrow e_3 \ddagger e_2$$

Proof.

$$\begin{aligned}
& e_1 \ddagger e_2 \wedge e_3 \sqsubseteq e_1 \\
& \quad \Rightarrow A_1 \cap \mathcal{T}(e_2) = \emptyset \wedge A_2 \cap \mathcal{T}(e_1) = \emptyset \wedge e_3 \sqsubseteq e_1 && \text{(def. of } \ddagger) \\
& \quad \Rightarrow \begin{pmatrix} A_1 \cap \mathcal{T}(e_2) = \emptyset \wedge A_2 \cap \mathcal{T}(e_1) = \emptyset \wedge \\ A_3 \subseteq A_{e_1} \wedge \mathcal{T}(e_3) \subseteq \mathcal{T}(e_1) \end{pmatrix} && \text{(def. of } \sqsubseteq) \\
& \quad \Rightarrow A_3 \cap \mathcal{T}(e_2) = \emptyset \wedge A_2 \cap \mathcal{T}(e_3) = \emptyset && \text{(appl. of } \subseteq\text{, substitution)} \\
& \quad \Rightarrow e_3 \ddagger e_2 && \text{(def. of } \ddagger)
\end{aligned}$$

□

Lemma 3.1.9. *For effects* $e_1 = (A_1, P_1)$, $e_1 = (A_2, P_2)$,

$$e_1 \sqsubseteq e_2 \Leftrightarrow \mathcal{R}(e_1) \sqsubseteq \mathcal{R}(e_2)$$

Proof. Let $\mathcal{R}(e_1) = (A_{\mathcal{R}_1}, P_{\mathcal{R}_1})$ and $\mathcal{R}(e_2) = (A_{\mathcal{R}_2}, P_{\mathcal{R}_2})$. Then by the definition of $\mathcal{R}$ and lemma 3.1.1, we have:

$$(A_{\mathcal{R}_1} = A_1) \wedge (A_{\mathcal{R}_2} = A_2) \wedge \big(\mathcal{T}(\mathcal{R}(e_1)) = \mathcal{T}(e_1)\big) \wedge \big(\mathcal{T}(\mathcal{R}(e_2)) = \mathcal{T}(e_2)\big).$$

$$\begin{aligned}
& e_1 \sqsubseteq e_2 \\
& \quad \Leftrightarrow A_1 \subseteq A_2 \wedge \mathcal{T}(e_1) \subseteq \mathcal{T}(e_2) && \text{(def. of } \sqsubseteq) \\
& \quad \Leftrightarrow A_{\mathcal{R}_1} \subseteq A_{\mathcal{R}_2} \wedge \mathcal{T}(e_1) \subseteq \mathcal{T}(e_2) && \text{(substitution)} \\
& \quad \Leftrightarrow A_{\mathcal{R}_1} \subseteq A_{\mathcal{R}_2} \wedge \mathcal{T}\big(\mathcal{R}(e_1)\big) \subseteq \mathcal{T}\big(\mathcal{R}(e_2)\big) && \text{(substitution)} \\
& \quad \Leftrightarrow \mathcal{R}(e_1) \sqsubseteq \mathcal{R}(e_2) && \text{(def. of } \mathcal{R})
\end{aligned}$$

□

Lemma 3.1.10. *For a function $f : D \to R$, let $F : \wp(D) \to \wp(R)$ be defined as $F(X) = \{f(x) : x \in X\}$, the element-wise application of f to X, a subset of D. Then for all A_1, P_1, A_2, P_2 such that $A_1, P_1, A_2, P_2 \subseteq D$ and all $f : D \to R$ for some R,*

$$\big(F(A_1), F(P_1)\big) \ddagger \big(F(A_2), F(P_2)\big) \Rightarrow (A_1, P_1) \ddagger (A_2, P_2).$$

Proof. We show the contrapositive, *i.e.*,

$$\neg\big((A_1, P_1) \ddagger (A_2, P_2)\big) \Rightarrow \neg\big(\big(F(A_1), F(P_1)\big) \ddagger \big(F(A_2), F(P_2)\big)\big).$$

$$\begin{aligned}
&\neg\big(\big((A_1, P_1) \ddagger (A_2, P_2)\big)\\
&\quad \Rightarrow \exists x : x \in \mathfrak{T}\big((A_1, P_1)\big) \wedge x \in A_2 && \text{(def. of } \ddagger\text{, appl. of } \neg)\\
&\quad \Rightarrow \exists x : x \in \mathfrak{T}\big((A_1, P_1)\big) \wedge x \in A_2 \wedge f(x) \in \mathfrak{T}\big((F(A_1), F(P_1))\big) \wedge f(x) \in F(A_2)\\
&&& \text{(def. of } F)\\
&\quad \Rightarrow \neg\big((F(A_1), F(P_1)) \ddagger (F(A_2), F(P_2))\big) && \text{(def. of } \ddagger)
\end{aligned}$$

Therefore, $\big(F(A_1), F(P_1)\big) \ddagger \big(F(A_2), F(P_2)\big) \Rightarrow \big((A_1, P_1) \ddagger (A_2, P_2)\big)$. □

Remark. Observe that converse of the conditional in lemma 3.1.10 is *not* true because when $(A_1, P_1) \ddagger (A_2, P_2)$ there might be some x, y such that $x \neq y \wedge x \in A_1 \wedge y \in A_2$ but $f(x) = f(y)$, and thus that $F(A_1) \cap F(A_2) \neq \emptyset$, so $\neg\big((F(A_1), F(P_1)) \ddagger (F(A_2), F(P_2))\big)$.

3.2 Understanding Effects in Context

Effects are not particularly interesting without context. However, when combined with a formal description of programs, they enable the proof of non-trivial properties of parallel programs, such as the fact that statements with non-interfering effects commute (theorem 4 in section 3.4).

Notation

While most of the notation used here is standard, there are a few notable exceptions.

- The $\in$ ("is element of") relation is overloaded to be meaningful for sequences as well as sets, in the obvious way.
- Expressions such as $x.T$ refer to the component of tuple x named T in the definition of x.
- The notation $[x : T]$ is taken to mean "the type associated with object x is T," *i.e.*, $x.T = T$.
- A parenthesized zero-subscript such as in $\sigma_{(0)}$ is taken to mean "with every occurence of a variable x replaced with x_0".
- $[\mathit{seq}_1 \rightarrow \mathit{seq}_2]$ is taken to mean "with each occurence of a variable in seq_1 replaced with the corresponding variable in seq_2".
- $\langle a_1, a_2, \ldots \rangle$ denotes an ordered sequence, and $\circ$ is sequence concatenation.
- $\sigma \xrightarrow{s} \sigma'$ means roughly "statement s takes state σ to state σ'". It is formally defined in eq. (3.3) on page 53.

3.2.1 Definitions

Definition 3.2.1 (Object)**.** An *object* $x = (T, R)$ is a tuple consisting of a type T and a realization R.

Definition 3.2.2 (Type)**.** A *type* $T = (V, v_0, \mathit{Op})$ is a tuple consisting of a (possibly infinite) set of values V, initial value $v_0 \in V$, and a sequence Op of operation contracts.

Definition 3.2.3 (State)**.** A *state* for a universe of objects X is a function $\sigma : X \to \big(\bigcup T, R : (T, R) \in X : T.V\big)$ such that each object $x = (T, R)$ has a value $\sigma(x) \in T.V$.

Definition 3.2.4 (State Space)**.** The *state space* $\hat{\sigma}(X)$ for a universe of objects X is the set of all possible states on the objects in X.

Definition 3.2.5 (Partition)**.** A *partition* of a universe of objects X is a function $\mathcal{P}$ from X to nonempty finite sets.

Definition 3.2.6 (Collective Partition)**.** The *collective partition* $\hat{\mathcal{P}}(X)$ of a universe of objects X is

$$\hat{\mathcal{P}}(X) = \bigcup x : x \in X : \mathcal{P}(x).$$

Definition 3.2.7 (Operation Contract)**.** An *operation contract* $o = (i, \pi, \mathit{pre}, \mathit{post}, \mathcal{S})$ is a tuple consisting of an identifier i, a sequence of parameters (*i.e.*, objects) π, a precondition predicate *pre*, a postcondition predicate *post*, and a specified effect $\mathcal{S}$.

Definition 3.2.8 (Specified Effect)**.** For an operation contract $o = (i, \pi, \mathit{pre}, \mathit{post}, \mathcal{S})$, the *specified effect* is a function $\mathcal{S} : \hat{\sigma}(\pi) \to \big\{(A, P) : \mathcal{T}\big((A, P)\big) \subseteq \hat{\mathcal{P}}(\pi)\big\}$.

Definition 3.2.9 (Realization)**.** The *realization* of object $x = (T, R)$ is a tuple $R = (B, F, I, C, \mathcal{J})$ consisting of a sequence B of operation bodies (*i.e.*, sequences of program statements), a set F of fields (*i.e.*, objects), a representation invariant $I : \hat{\sigma}(F) \to \{\mathsf{true}, \mathsf{false}\}$, an abstraction relation $C \subseteq \hat{\sigma}(F) \times T.V$, and a function $\mathcal{J} : \hat{\sigma}(F) \times \hat{\mathcal{P}}(F) \to \mathcal{P}(x)$.

Definition 3.2.10 (Actual Effect)**.** The *actual effect* of an operation body b with parameters π for some $\sigma \in \hat{\sigma}(\pi)$, denoted $\mathcal{A}_\sigma(b)$, is the combined effect of each statement in b that is executed when b starts with its parameters having the values in σ. Details of its structure are below, in section 3.2.3.

Definition 3.2.11 (The Implements relation)**.** An operation body b *implements* an operation contract $o = (i, \pi, \mathit{pre}, \mathit{post}, \mathcal{S})$, denoted $b \mapsto o$, if

$$\forall \sigma, \sigma' : \sigma \xrightarrow{b} \sigma' \Rightarrow \sigma \xrightarrow{o(\pi)} \sigma'$$

A realization $R = (B, F, I, C, \mathfrak{I})$ *implements* a type $T = (V, v_0, \mathit{Op})$, denoted $R \mapsto T$, if

$$\forall i : 0 \leq i < |\mathit{Op}| : B_i \mapsto \mathit{Op}_i.$$

Definition 3.2.12 (The Respects relation)**.** An operation body b *respects* an operation contract $o = (i, \pi, \mathit{pre}, \mathit{post}, \mathcal{S})$, denoted $b \rightarrowtail o$, if

$$\forall \sigma : \sigma \vdash \mathit{pre} : \mathcal{A}_\sigma(b) \sqsubseteq \mathcal{S}(\sigma).$$

A realization $R = (B, F, I, C, \mathfrak{I})$ *respects* a type $T = (V, v_0, \mathit{Op})$, denoted $R \rightarrowtail T$, if

$$\forall i : 0 \leq i < |\mathit{Op}| : B_i \rightarrowtail \mathit{Op}_i.$$

Definition 3.2.13 (Validity)**.** An operation body b is *valid* for an operation contract o if $b \mapsto o \wedge b \rightarrowtail o$.

A realization R is *valid* for a type T if every operation body is valid for the corresponding contract in T.

3.2.2 Well-Formedness Conditions

Definition 3.2.14 (Well-formedness of specified effects)**.** A specified effect $\mathcal{S}$ is *well-formed* if $\big(\forall \sigma : \mathcal{S}(\sigma)$ is well-formed$\big)$. $\mathcal{S}$ is *well-formed with respect to* a set T if $\mathcal{S}$ is well-formed and $\big(\forall \sigma : \mathcal{T}(\mathcal{S}(\sigma)) \subseteq T\big)$.

Definition 3.2.15 (Well-formedness of operation contracts)**.** An operation contract $o = (i, \pi, \mathit{pre}, \mathit{post}, \mathcal{S})$ is *well-formed* if:

1. Every identifier in π is unique
2. Every free variable in *pre* is in π
3. Every free variable in *post* is of the form x or x_0 where x is in π
4. $\mathcal{S}$ is well-formed with respect to $\hat{\mathcal{P}}(\pi)$

Definition 3.2.16 (Well-formedness of types)**.** A type $T = (V, v_0, Op)$ is *well-formed* if each contract in Op is well-formed.

Definition 3.2.17 (Well-formedness of objects)**.** An object $x = (T, R)$ is *well-formed* if T is well-formed and R is valid for T.

Hereafter, we are concerned only with well-formed objects and operation contracts.

3.2.3 Programs

The above definitions amount to a model of programming on which a language can be built. A program in this model consists of a sequence of *statements*. Each statement is a control structure (*e.g.*, a conditional statement or a loop) or an operation call that has some number of arguments that correspond to that operation's parameters. Each statement has a *behavior* and an *effect*. For an operation call, the behavior of the statement is some relation on the values of the arguments, derived from the behavioral specification (*e.g.*, pre- and post-conditions).[2] Effects are derived from a non-interference specification (part of an operation contract) and are manipulated via the A/P Calculus introduced in section 3.1. An operation is implemented by an operation body, a sequence of statements involving the parameters of the operation.

[2]Behavioral specifications and verification are discussed at length in the literature [68, 79, 115, 106, 110]; it is assumed that there is a mechanism by which a program may be reasoned about, that is, there is a definition of correctness and a formal semantics.

3.3 The Language

We define a simple programming language (along with its semantics with respect to effects) that enables the computation of the actual effect of an operation body beginning in a particular state. The grammar for the language is in fig. 3.8.

$\langle\mathit{body}\rangle$	$::=$	$\texttt{operation}\ \langle\mathit{op\text{-}name}\rangle\,\texttt{(}\langle\mathit{id\text{-}list}\rangle\texttt{)}\ \texttt{:}\ \langle\mathit{stmt}\rangle\ \texttt{end}$	Operation Body
$\langle\mathit{id\text{-}list}\rangle$	$::=$	$\langle\mathit{id}\rangle\texttt{,}\ \langle\mathit{id\text{-}list}\rangle$	Identifier List
	$\mid$	$\langle\mathit{id}\rangle$	Identifier
$\langle\mathit{stmt}\rangle$	$::=$	ε	Empty Statement
	$\mid$	$\langle\mathit{simp\text{-}stmt}\rangle\texttt{;}$	Simple Statement
	$\mid$	$\langle\mathit{stmt}\rangle_1\ \langle\mathit{stmt}\rangle_2$	Sequential Composition
	$\mid$	$\texttt{if}\ \langle\mathit{id}\rangle\ \texttt{then}\ \langle\mathit{stmt}\rangle_1\ \texttt{else}\ \langle\mathit{stmt}\rangle_2\ \texttt{end}$	If Statement
	$\mid$	$\texttt{while}\ \langle\mathit{id}\rangle\ \texttt{do}\ \langle\mathit{stmt}\rangle\ \texttt{end}$	While Statement
	$\mid$	$\texttt{cobegin}\ \langle\mathit{par\text{-}block}\rangle\ \texttt{end}$	Cobegin Statement
$\langle\mathit{simp\text{-}stmt}\rangle$	$::=$	$\langle\mathit{type\text{-}id}\rangle\ \langle\mathit{id}\rangle$	Variable Introduction
	$\mid$	$\langle\mathit{id}\rangle_1 :=: \langle\mathit{id}\rangle_2$	Swap
	$\mid$	$\langle\mathit{op\text{-}call}\rangle$	
$\langle\mathit{op\text{-}call}\rangle$	$::=$	$\langle\mathit{op\text{-}name}\rangle\ \texttt{(}\langle\mathit{id\text{-}list}\rangle\texttt{)}$	Operation Call
$\langle\mathit{par\text{-}block}\rangle$	$::=$	$\langle\mathit{op\text{-}call}\rangle \parallel \langle\mathit{par\text{-}block}\rangle$	Parallel Block
	$\mid$	$\langle\mathit{op\text{-}call}\rangle$	

Figure 3.8: Context-free grammar for our programming language.

It is important to note that this language is just one possible programming language that implements the programming model from section 3.2. There is nothing in this work that fundamentally requires this particular language, although semantics are—in this chapter—defined concretely in its terms. (In fact subsequent chapters make use of a fully-fledged language called RESOLVE [110, 62, 45, 126] to implement these ideas.)

One nonstandard restriction placed on the structure of programs in a language that implements the programming model from section 3.2 is that an $\langle$*id-list*$\rangle$ must consist of *unique* $\langle$*id*$\rangle$s. This restriction is left implicit in the inference rules below to improve readability.

3.3.1 Parameter Passing and Aliases

The language above, by design, does not premit aliases in the usual sense although parameter passing in the language is, strictly speaking, by reference. However, parameter passing by reference might be said to introduce an alias (between the argument and the formal parameter) even though the two names are not in scope at the same time. In sequential programs, this cannot introduce unsoundness to reasoning with value semantics (that is, as if there were no references). Unsoundness can arise, however, when concurrency is introduced such as by a **`cobegin`** statement in which several parallel operation calls share an argument. If several parallel operation calls attempt to mutate the same object, it could produce an inconsistent state that does not satisfy the representation invariant of that object. A consequence of the results proven below in section 3.4 is that given pairwise non-interference between the specified effects of $\langle$*op-call*$\rangle$s in a $\langle$*par-block*$\rangle$, the value of an object is always well-defined even in the presence of such data sharing among threads.

It is due to this tension between reasoning about sequential and concurrent programs that the language has two *conceptual* parameter passing mechanisms. Arguments to operation calls in **`cobegin`** statements are reasoned about using pass-by-reference semantics, while arguments to all other operation calls use pass-by-swapping semantics. Under pass-by-swapping, each argument is swapped into the corresponding formal parameter at the point of the call—leaving the argument with an initial value for its type—and that formal parameter is swapped back to the argument at the end of the operation. Such a reasoning "shortcut" guarantees that in a sequential program there is *never* a time at which there are two names for the same object—even names that are not simultaneously in scope. Pass-by-swapping also sidesteps the well-known repeated arguments problem [92] because it provides for well-defined semantics when the same argument is provided several times to the same operation call.

While alias freedom may seem like a too-restrictive condition to place on programs, it has been shown that real-world software can be built entirely without aliases [81] and when they are absolutely necessary, aliases may be dealt with as a special case [132]. Several popular modern programming languages, in fact, have mechanisms to enable alias-free programming [89, 78] and can require the programmer to put in extra work to introduce aliases. In our programming model, the alias-free restriction manifests in the following axiom.

Axiom 1 (Alias Freedom)**.** *For any two distinct objects x, y in scope, $\mathcal{P}(x) \cap \mathcal{P}(y) = \emptyset$.*

3.3.2 Primitive Operations

Without additional machinery, programs in the model of section 3.2 have no lowest-level implementation because everything is implemented in terms of something else. Therefore, it is necessary to ground these programs with primitive operations that do not themselves have operation bodies. Each primitive operation is *atomic*: it occurs in one "step" of execution and it operates on entire objects rather than pieces of objects. A primitive operation in any language that implements our programming model is a refinement of the contract below for `Prim`.

$$\begin{pmatrix} i : \texttt{Prim}, \\ \pi : \pi_A \circ \pi_P, \\ pre : \textsf{true}, \\ post : \forall x : x \in \pi_P : x = x_0 \\ \mathcal{S}(\sigma) : \left(\hat{\mathcal{P}}(\pi_A), \hat{\mathcal{P}}(\pi_P)\right) \end{pmatrix} \qquad (\texttt{Prim})$$

3.3.3 Possible Primitive Operations

Although the `Prim` contract is quite general, there are two specific primitive operations that are used in the language of fig. 3.8. These operations are `Init` and `Swap`.

The Init Operation

The first primitive operation in the language is the `Init` operation. Its contract is below.

$$\begin{pmatrix} i : \texttt{Init}, \\ \pi : \langle T, x\rangle, \\ pre : \textsf{true}, \\ post : x = T.v_0, \\ \mathcal{S}(\sigma) : \big(\mathcal{P}(x), \emptyset\big) \end{pmatrix} \tag{Init}$$

The `Init` operation sets the value of its argument to the initial value for its type. It is used implicitly in variable introduction statements.

The Swap Operation

The other primitive operation is `Swap`, with the following contract.

$$\begin{pmatrix} i : \texttt{Swap}, \\ \pi : \langle x, y\rangle, \\ pre : \textsf{true}, \\ post : (x = y_0 \land y = x_0), \\ \mathcal{S}(\sigma) : \big(\mathcal{P}(x) \cup \mathcal{P}(y), \emptyset\big) \end{pmatrix} \tag{Swap}$$

The `Swap` operation is used implicitly by the :=: operator in the language.

3.3.4 Using the Language

Although the language of fig. 3.8 is essentially a toy language for showing proof of concept, it is reasonably complete in the sense that it can be used to write any program (although those programs may be quite verbose).

Example: Defining a Constant

A constant can be implemented within the language by defining a new type for each value a programmer wishes to use. For example, the integral constant 42 is implemented by the type C42 in eq. (3.1); it is instantiated by a variable introduction statement such as "`C42 FORTY_TWO;`".

$$\texttt{C42} = \big(\{42\}, 42, \langle\rangle\big) \tag{3.1}$$

Example: Implementing Bit

Using the primitive operations (`Swap`) and (`Init`), one could implement any other component. For example, it is quite easy to implement a single bit that provides two operations (`Set` and `Unset`, defined below) using the primitive operations in the language (listing 3.2). Objects of type `Bit` (eq. (3.2)) have a singleton partition, that is, $\forall b : [b : \texttt{Bit}] : \mathcal{P}(b) = \{b\}$.

$$\texttt{Bit} = \big(\{\mathsf{true}, \mathsf{false}\}, \mathsf{false}, \langle \texttt{Set}, \texttt{Unset}\rangle\big) \tag{3.2}$$

$$\begin{pmatrix} i : \texttt{Set}, \\ \pi : \langle b \rangle, \\ \mathit{pre} : [b : \texttt{Bit}], \\ \mathit{post} : b, \\ \mathcal{S}(\sigma) : \big(\{b\}, \emptyset\big) \end{pmatrix} \tag{Set}$$

$$\begin{pmatrix} i : \texttt{Unset}, \\ \pi : \langle b \rangle, \\ pre : [b : \texttt{Bit}], \\ post : \neg b, \\ \mathcal{S}(\sigma) : \big(\{b\}, \emptyset\big) \end{pmatrix} \tag{\texttt{Unset}}$$

The implementation begins by defining two constant types, `T_TYPE` and `F_TYPE` as follows.

$$\texttt{T\_TYPE} = \big(\{\mathsf{true}\}, \mathsf{true}, \langle\rangle\big)$$

$$\texttt{F\_TYPE} = \big(\{\mathsf{false}\}, \mathsf{false}, \langle\rangle\big)$$

The operation bodies for the two `Bit` operation contracts are in listing 3.2.

Listing 3.2: Valid operation bodies for (`Set`) and (`Unset`).

```
operation Set(b) :
  T_TYPE TRUE;
  b :=: TRUE;
end

operation Unset(b) :
  F_TYPE FALSE;
  b :=: FALSE;
end
```

Example: Layering Implementations

Observe that by using objects of type `Bit`, it is possible to implement a functionally complete set of logical operators (*e.g.*, by implementing NOR, denoted by $\downarrow$), and therefore to simulate a computer.[3] For example, the operation body in listing 3.3 is valid for the operation contract (`Nor`).

$$\begin{pmatrix} i : \texttt{Nor}, \\ \pi : \langle a, b\rangle, \\ pre : \big([a : \texttt{Bit}] \wedge [b : \texttt{Bit}]\big), \\ post : \big((a = a_0 \downarrow b_0) \wedge b = b_0\big), \\ \mathcal{S}(\sigma) : \big(\{a\}, \{b\}\big) \end{pmatrix} \tag{Nor}$$

Listing 3.3: Valid operation body for (`Nor`).

```
operation Nor(a, b) :
  if a then
    Unset(a);
  else
    if b then
      Unset(a);
    else
      Set(a);
    end
  end
end
```

3.3.5 Behavioral Semantics

In this section, the semantics of the language discussed in section 3.3 are formalized. The behavior of a statement s is notated with $\xrightarrow{s}$ (3.3), a (non-total) relation between states defined by $\mathsf{post}(s)$. The relation is restricted by $\mathsf{pre}(s)$—that is, $\sigma \xrightarrow{s} \sigma'$ is only defined when $\sigma \vdash \mathsf{pre}(s)$. For an operation call, $\mathsf{pre}(s)$ and $\mathsf{post}(s)$ are equal

[3]It is in this respect that we mean the programming model is "reasonably complete"—it can be used to simulate any other programming model, though no rigorous proof is given to this effect.

to the *pre* and *post* predicates in the contract for the operation.

$$\sigma \xrightarrow{s} \sigma' \Leftrightarrow \big(\sigma \vdash \mathsf{pre}(s) \wedge \sigma_{(0)}, \sigma' \vdash \mathsf{post}(s)\big) \tag{3.3}$$

Frame Rule

There is an enormous body of work on formalizing the semantics of programming languages, so most of the rules below (3.5–14) are not too interesting. One deviation from standard semantics is the formulation of the frame rule. Typically, a frame rule is formulated in terms of objects and their memory footprints; here it is defined in terms of partitioned objects and effects on those objects.

$$\textsc{Frame Rule}\qquad \frac{\sigma \xrightarrow{s} \sigma' \qquad \mathcal{A}_\sigma(s).A \cap \mathcal{P}(x) = \emptyset}{\sigma(x) = \sigma'(x)} \tag{3.4}$$

The rule for sequential composition (3.10) is slightly awkward because it accounts for relational behavior and the semantics of ⟨*par-block*⟩ in eq. (3.14) is perhaps surprising. The parallel composition of two statements is modeled as an arbitrary interleaving of the constituent instructions within those statements. Informally, the rule for ⟨*par-block*⟩ states that whenever the operation calls in ⟨*par-block*⟩ are non-interfering, if the precondition of *any* permutation of the calls in ⟨*par-block*⟩ is satisfied by state σ then the resultant state σ' satisfies the postcondition of *every* permutation of the calls in ⟨*par-block*⟩.[4] The soundness of these semantics is a consequence of the results from section 3.4.

The "context" or "environment" for the rules below is provided as a pair (X, M), a symbol table consisting of X, the objects in scope, and M, the operation contracts

[4]Strictly speaking, the rule as written only considers permutations where ⟨*op-call*⟩ is the first or last call in the sequence. This does not compromise soundness, and is notationally convenient. Theorem 4 does not restrict permutations in this way.

in scope. The context is changed only by eq. (3.7), in which X grows to X' (in all other rules $X = X'$). Finally, $\sigma \in \hat{\sigma}(X)$ and $\sigma' \in \hat{\sigma}(X')$.

Semantics of Operation Body

$$\frac{b = \text{operation } Op(ids)\text{: } s \text{ end} \qquad \sigma \vdash pre \qquad \sigma \xrightarrow{s} \sigma'}{\sigma \xrightarrow{b} \sigma'} \tag{3.5}$$

Semantics of ε

$$\frac{}{\sigma \xrightarrow{\varepsilon} \sigma} \tag{3.6}$$

Semantics of Variable Introdudction

$$\frac{\sigma = \sigma' \restriction_X \qquad X \cup \{x\} = X' \qquad [x : T] \qquad \sigma'(x) = T.v_0}{\sigma \xrightarrow{T\ x;} \sigma'} \tag{3.7}$$

Semantics of Swap

$$\frac{\sigma = \sigma'[\langle x, y\rangle \to \langle y, x\rangle]}{\sigma \xrightarrow{x:=:y} \sigma'} \tag{3.8}$$

Semantics of Operation Call

$$\frac{\big(Op, \pi, pre, post, \mathcal{S}\big) \in M \qquad \sigma \vdash pre[\pi \to ids] \qquad \sigma_{(0)}, \sigma' \vdash post[\pi \to ids]}{\sigma \xrightarrow{Op(ids);} \sigma'} \tag{3.9}$$

Semantics of Sequential Composition

$$\frac{\forall \sigma' : \sigma \xrightarrow{s_1;} \sigma' : \sigma' \vdash \mathsf{pre}(s_2) \qquad \forall \sigma' : \sigma' \xrightarrow{s_2;} \sigma'' : \sigma_{(0)}, \sigma' \vdash \mathsf{post}(s_1)}{\sigma \xrightarrow{s_1;s_2;} \sigma''} \tag{3.10}$$

Semantics of If Statement

$$\frac{\sigma(x) \Rightarrow \sigma \xrightarrow{s_1;} \sigma' \qquad \neg\sigma(x) \Rightarrow \sigma \xrightarrow{s_2;} \sigma'}{\sigma \xrightarrow{\text{if } x \text{ then } s_1 \text{ else } s_2 \text{ end;}} \sigma'} \tag{3.11}$$

Semantics of While Statement

$$\frac{\sigma(x) \Rightarrow \sigma \xrightarrow{s_1;\ \text{while } x \text{ do } s_1 \text{ end;}} \sigma' \qquad \neg\sigma(x) \Rightarrow \sigma \xrightarrow{\varepsilon} \sigma'}{\sigma \xrightarrow{\text{while } x \text{ do } s_1 \text{ end;}} \sigma'} \tag{3.12}$$

Semantics of Cobegin

$$\frac{\sigma \xrightarrow{par} \sigma'}{\sigma \xrightarrow{\text{cobegin } par \text{ end;}} \sigma'} \tag{3.13}$$

Semantics of Parallel Block

$$\frac{\sigma \vdash \big(\mathsf{pre}(Op; par;) \vee \mathsf{pre}(par; Op;)\big) \qquad \overset{\mathcal{A}_\sigma(Op) \ddagger \mathcal{A}_\sigma(par)}{\sigma_{(0)}, \sigma' \vdash \big(\mathsf{post}(Op; par;) \wedge \mathsf{post}(par; Op;)\big)}}{\sigma \xrightarrow{Op\|par} \sigma'} \tag{3.14}$$

3.3.6 Defining the Effect of a Program

The actual effect of a statement ($\mathcal{A}_\sigma(s)$) is defined for each kind of statement in the language. Any agent that determines the actual effect of a statement must do so in tandem with the determination of the behavior of that statement: the actual effect depends on the values of the objects as derived from the behavioral semantics in section 3.3.5.

It is frequently useful to apply the non-interference correspondence, $\mathcal{I}$, of a realization $R = (B, F, I, C, \mathcal{I})$ to an entire effect e (rather than to a single piece of a partition). For this purpose the related function $\hat{\mathcal{I}} : \hat{\sigma}(F) \times E \to E$ is defined as follows, where $E = \{e : e \text{ is a well-formed effect}\}$.

$$\hat{\mathcal{I}}(\sigma, e) = \begin{array}{c} \big(\{\mathcal{I}(\sigma, p) : p \in e|_{\hat{\mathcal{P}}(F)}.A\}, \{\mathcal{I}(\sigma, p) : p \in e|_{\hat{\mathcal{P}}(F)}.P\}\big) \sqcup \\ \big(\{p : p \in (e.A \setminus \hat{\mathcal{P}}(F))\}, \{p : p \in (e.P \setminus \hat{\mathcal{P}}(F))\}\big) \end{array} \tag{3.15}$$

The actual effect of an operation body is defined by eq. (3.16) below. For that rule, realization $R = (B, F, I, C, \mathcal{I})$ is well-formed with respect to a specification $T = (V, v_0, Op)$, operation contract $o = (i, \pi, pre, post, \mathcal{S})$ is such that $o \in O$, and $\sigma \in \hat{\sigma}(\pi)$.

The defintions of the actual effect for the other kinds of statements and expressions are in eqs. (3.17–25) below. For each equation below, we are given $S = (X, M)$ and $\sigma \in \hat{\sigma}(X)$ (as in section 3.3.5).

- Effect of Operation Body: $b =$ operation $Op(ids)$: s end

$$\mathcal{A}_\sigma(b) = \hat{\mathfrak{I}}\big(\sigma, \mathcal{A}_\sigma(s)\big)|_{\hat{\mathcal{P}}(ids)} \tag{3.16}$$

- Effect of Empty Statement: $s = \ \varepsilon$

$$\mathcal{A}_\sigma(s) = \bot \tag{3.17}$$

- Effect of Variable Introduction: $s = \ T\ x;$

$$\mathcal{A}_\sigma(s) = \big(\mathcal{P}(x), \emptyset\big) \tag{3.18}$$

- Effect of Swap Statement: $s = \ x :=: y;$

$$\mathcal{A}_\sigma(s) = \big(\mathcal{P}(x) \cup \mathcal{P}(y), \emptyset\big) \tag{3.19}$$

- Effect of Operation Call: $s = \ Op(ids)$ where $(Op, \pi, pre, post, \mathcal{S})$ is the well-formed contract of some operation.

$$\mathcal{A}_\sigma\big(s\big) = \mathcal{S}(\sigma[ids \to \pi])[\pi \to ids] \tag{3.20}$$

- Effect of Sequential Composition: $s = \ s_1; s_2$

$$\mathcal{A}_\sigma(s) = \mathcal{A}_\sigma(s_1) \sqcup \Big(\bigsqcup \sigma' : \sigma \xrightarrow{s_1} \sigma' : \mathcal{A}_{\sigma'}(s_2)\Big) \tag{3.21}$$

- Effect of If Statement: $s = \ $ if x then s_1 else s_2 end

$$\mathcal{A}_\sigma(s) = \big(\emptyset, \mathcal{P}(x)\big) \sqcup \begin{cases} \mathcal{A}_\sigma(s_1)|_{\hat{\mathcal{P}}(X)} & \sigma(x) \\ \mathcal{A}_\sigma(s_2)|_{\hat{\mathcal{P}}(X)} & \neg\sigma(x) \end{cases} \tag{3.22}$$

- Effect of While Statement: $s = \ $ while x do s_1 end

$$\mathcal{A}_\sigma(s) = \big(\emptyset, \mathcal{P}(x)\big) \sqcup \begin{cases} \mathcal{A}_\sigma(s_1; s)|_{\hat{\mathcal{P}}(X)} & \sigma(x) \\ \bot & \neg\sigma(x) \end{cases} \tag{3.23}$$

- Effect of Cobegin Statement: $s =$ cobegin *par* end

$$\mathcal{A}_\sigma(s) = \mathcal{A}_\sigma(par) \tag{3.24}$$

- Effect of Parallel Block: $s = \ Op \| par$

$$\mathcal{A}_\sigma(s) = \mathcal{A}_\sigma(Op) \sqcup \mathcal{A}_\sigma(par) \tag{3.25}$$

3.4 Results

Through the combination of the A/P Calculus of section 3.1 and the programming model and language of sections 3.2 and 3.3, it can be shown that non-interfering statements always commute. These results have implications for reasoning about and verifiying the correctness of parallel programs in languages that implement the programming model described above.

Lemma 3.4.1 states that any object not mentioned in a statement is not in the target of the actual effect of that statement; by the frame rule, then, any object not mentioned in a statement does not have its value changed by that statement. It is useful in applying the frame rule even to objects that are not mentioned in a statement.

Lemma 3.4.1. *For any statement s, state σ, and object x not mentioned in s, $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.*

Proof. We proceed by induction on s.

Base Cases

- $s = \ \varepsilon$ (empty statement).

  By eq. (3.17), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \emptyset$ and the lemma is trivially true.

- $s =$ $T\ y$ (variable introduction).

  By eq. (3.18), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y)$. If x is not mentioned in s, then $x \neq y$. Therefore, $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

- $s =$ $y :=: z$ (swap statement).

  By eq. (3.19), $\mathcal{T}(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y) \cup \mathcal{P}(z)$. It x is not mentioned in s, then $x \neq y$ and $x \neq z$. Therefore, $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

- $s =$ $Op(ids)$ (operation call).

  By well-formedness definitions 3.2.14 and 3.2.15 and eq. (3.20), if x is not mentioned in s then $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

Inductive Step

For induction, assume that s_1 and s_2 are such that x does not appear in either statement (and thus have the property that $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big) \cap \mathcal{P}(x) = \emptyset$).

- $s =$ $s_1; s_2$.

  By eq. (3.21), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big)$. By inductive hypothesis, $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big) \cap \mathcal{P}(x) = \emptyset$. Thus, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

- $s =$ if y then s_1 else s_2 end.

By eq. (3.22), either $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big)$ or $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big)$. If x is not mentioned in x, then $y \neq x$; by inductive hypothesis $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big) \cap \mathcal{P}(x) = \emptyset$. Thus, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

- $s =$ while y do s_1 end.

  By eq. (3.23), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big)$. If x is not mentioned in s, then $x \neq y$; by inductive hypothesis $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$. Thus, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

- $s =$ cobegin $Op \| par$ end.

  By eq. (3.25), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{T}\big(\mathcal{A}_\sigma(Op)\big) \cup \mathcal{T}\big(\mathcal{A}_\sigma(par)\big)$. Since by assumption $\mathcal{T}\big(\mathcal{A}_\sigma(Op)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(par)\big) \cap \mathcal{P}(x) = \emptyset$, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

$\square$

Lemma 3.4.2 states that the behavior of a statement depends only upon variables mentioned in that statement. It serves as a justification for ignoring overly-broad specifications when reasoning about non-interference.

Let $\hat{X}_\sigma(s) = \big\{x : \mathcal{P}(x) \cap \mathcal{T}(\mathcal{A}_\sigma(s)) \neq \emptyset\big\}$ (*i.e.*, the objects mentioned in s in a statement that is executed when s begins in state σ). $\bar{X}_\sigma(s)$ is its complement. $f\restriction_S$ denotes the restriction of function $f : D \to R$ to the domain $S \cap D$.

Lemma 3.4.2. *For all statements s and states σ_1, σ_1', σ_2, and σ_2',*

$$\left(\begin{array}{l} \left(\sigma_1 \restriction_{\hat{X}_{\sigma_1}(s)} = \sigma_2 \restriction_{\hat{X}_{\sigma_2}(s)}\right) \land \left(\sigma_1' \restriction_{\hat{X}_{\sigma_1}(s)} = \sigma_2' \restriction_{\hat{X}_{\sigma_2}(s)}\right) \ \land \\ \left(\sigma_1 \restriction_{\bar{X}_{\sigma_1}(s)} = \sigma_1' \restriction_{\bar{X}_{\sigma_1}(s)}\right) \land \left(\sigma_2 \restriction_{\bar{X}_{\sigma_2}(s)} = \sigma_2' \restriction_{\bar{X}_{\sigma_2}(s)}\right) \end{array}\right) \Rightarrow$$

$$\left((\sigma_1 \xrightarrow{s} \sigma_1') \Leftrightarrow (\sigma_2 \xrightarrow{s} \sigma_2')\right).$$

That is, whenever σ_1, σ_1', σ_2, and σ_2' are related as in fig. 3.9, it follows that $\sigma_1 \xrightarrow{s} \sigma_1' \Leftrightarrow \sigma_2 \xrightarrow{s} \sigma_2'$. An edge $\overset{D}{=\!=}$ between two states in the figure indicates they are equal when restricted to the domain D.

$$\begin{tikzcd} \sigma_1 \arrow[r, equal, "\bar{X}_{\sigma_1}(s)"] \arrow[d, equal, "\hat{X}_{\sigma_1}(s)"'] & \sigma_1' \arrow[d, equal, "\hat{X}_{\sigma_1}(s)"] \\ \sigma_2 \arrow[r, equal, "\bar{X}_{\sigma_2}(s)"] & \sigma_2' \end{tikzcd}$$

Figure 3.9: Diagram of the relationships between σ_1, σ_1', σ_2, and σ_2' described in lemma 3.4.2.

Proof. First, observe that for the antecedent to hold, it *must* be the case that $\hat{X}_{\sigma_1}(s) = \hat{X}_{\sigma_2}(s)$ (and, therefore that $\bar{X}_{\sigma_1}(s) = \bar{X}_{\sigma_2}(s)$). We can therefore reason (without loss of generality) only with $\hat{X}_{\sigma_1}(s)$ and $\bar{X}_{\sigma_1}(s)$.

We proceed by induction on s.

Base Cases

- $s = \varepsilon$ (empty statement).

  By eq. (3.17), $\mathcal{T}(\mathcal{A}_{\sigma_1}(s)) = \emptyset$; therefore $\hat{X}_{\sigma_1}(s) = \emptyset$, so $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$. By eq. (3.6), for any σ, $\sigma \xrightarrow{\varepsilon} \sigma$, so the lemma holds.

- $s = \ T\ x$ (variable introduction).

  By eq. (3.18), $\mathcal{T}(\mathcal{A}_{\sigma_1}(s)) = \mathcal{P}(x)$; therefore $\hat{X}_{\sigma_1}(s) = \{x\}$. By eq. (3.7), $\sigma_1 \xrightarrow{s} \sigma_1'$ whenever $\sigma_1 = \sigma_1' \restriction_X$ (where X is the set of in-scope objects before s) and σ_1' is additionally defined on x. Moreover, whenever $\sigma_1 \xrightarrow{s} \sigma_1'$, and the antecedent of the lemma holds, it is also true that $\sigma_2 \xrightarrow{s} \sigma_2'$. Therefore the lemma holds for $s = \ T\ x;$.

- $s = \ x :=: y$ (swap statement).

  By eq. (3.19), By eq. (3.19), $\mathcal{T}(\mathcal{A}_{\sigma_1}(s)) = \mathcal{P}(x) \cup \mathcal{P}(y)$; therefore $\hat{X}_{\sigma_1}(s) = \{x, y\}$. If $\sigma_1 \restriction_X = \sigma_2 \restriction_X$, then $\sigma_1(x) = \sigma_2(x)$ (analogously for y). By eq. (3.8), the only variables that have their values changed by s are x and y; therefore whenever $\sigma_1 \xrightarrow{s} \sigma_1'$, and the antecedent of the lemma holds, it is also true that $\sigma_2 \xrightarrow{s} \sigma_2'$. Therefore the lemma holds for $s = \ x :=: y$.

- $s = \ Op(ids)$ (operation call).

  Let there be some $(Op, \pi, pre, post, \mathcal{S}) \in M$. It follows from eq. (3.9) and the definition of $\xrightarrow{s}$ (3.3) that $\sigma_1 \vdash pre \Leftrightarrow \sigma_1 \vdash \mathsf{pre}(s)$. By assumption, for each object x mentioned in s, $\sigma_1(x) = \sigma_2(x)$. Because pre mentions only objects in π (by well-formedness definition 3.2.15), it follows that $\sigma_1 \vdash pre \Leftrightarrow \sigma_2 \vdash pre$ (and, therefore, that $\sigma_2 \vdash pre \Leftrightarrow \sigma_2 \vdash \mathsf{pre}(s)$). Therefore $\sigma_1 \vdash \mathsf{pre}(s) \Leftrightarrow \sigma_2 \vdash \mathsf{pre}(s)$. By analogous reasoning, we see that $\sigma_{1(0)}, \sigma_1' \vdash \mathsf{post}(s) \Leftrightarrow \sigma_{2(0)}, \sigma_2' \vdash \mathsf{post}(s)$. Therefore, $\sigma_1 \xrightarrow{s} \sigma_1' \Leftrightarrow \sigma_2 \xrightarrow{s} \sigma_2'$.

Inductive Step

Assume the lemma holds for s_1 and s_2.

- $s =$ $s_1; s_2$.

  Assume we have some states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ related as in fig. 3.9. Without loss of generality we proceed from $\sigma_1 \xrightarrow{s} \sigma_1'$. By eq. (3.10), $\sigma_1 \xrightarrow{s} \sigma_1'$ whenever

  $$\left(\forall \sigma'' : \sigma_1 \xrightarrow{s_1} \sigma'' : \sigma'' \vdash \mathsf{pre}(s_2)\right) \wedge \left(\forall \sigma'' : \sigma'' \xrightarrow{s_2} \sigma_1' : \sigma_{1(0)}, \sigma'' \vdash \mathsf{post}(s_1)\right).$$

  By our inductive hypothesis, then, it is also true that

  $$\left(\forall \sigma'' : \sigma_2 \xrightarrow{s_1} \sigma'' : \sigma'' \vdash \mathsf{pre}(s_2)\right) \wedge \left(\forall \sigma'' : \sigma'' \xrightarrow{s_2} \sigma_2' : \sigma_{2(0)}, \sigma'' \vdash \mathsf{post}(s_1)\right).$$

  Therefore, by eq. (3.10), $\sigma_2 \xrightarrow{s} \sigma_2'$. Therefore, given $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ related as in fig. 3.9, it follows that $\sigma_1 \xrightarrow{s} \sigma_1' \Leftrightarrow \sigma_1 \xrightarrow{s} \sigma_2'$.

- $s =$ if x then s_1 else s_2 end.

  By eq. (3.11), $\sigma_1 \xrightarrow{s} \sigma_1'$ if and only if either $\sigma_1 \xrightarrow{s_1} \sigma_1'$ or $\sigma_1 \xrightarrow{s_2} \sigma_1'$. Since by our inductive hypothesis the lemma holds for both s_1 and s_2, it also holds for s.

- $s =$ while x do s_1 end.

  By eq. (3.12), $\sigma_1 \xrightarrow{s} \sigma_1'$ if and only if either $\neg\sigma_1(x)$ (in which case $\sigma_1 = \sigma_1'$, and the lemma holds for the same reasons as when $s = \varepsilon$) or $\sigma_1(x)$ and $\sigma_1 \xrightarrow{s_1;\ \text{while } x \text{ do } s_1 \text{ end}} \sigma_1'$. Since we have shown that the lemma holds for $s_1; s_2$;, it also holds for s.

- $s =$ cobegin *par* end.

  By the grammar in fig. 3.8, *par* consists of a list of $\langle$*op-call*$\rangle$s o_i, which we showed above each individually satisfy the lemma. Therfore, if $\sigma_1 \vdash \mathsf{pre}(s)$, then by eq. (3.14) there must be some o_i in *par* such that $\sigma_1 \vdash \mathsf{pre}(o_i)$. So it is also true that $\sigma_2 \vdash \mathsf{pre}(o_i)$, so $\sigma_2 \vdash \mathsf{pre}(s)$. Similarly, if $\sigma_1' \vdash \mathsf{post}(s)$ then it follows that $\sigma_1' \vdash \mathsf{post}(o_i)$ for every o_i in *par*. Thus, we have that $\sigma_2' \vdash \mathsf{post}(o_i)$ for each i and therefore $\sigma_2' \vdash \mathsf{post}(s)$. Therefore the lemma holds for s.

Therefore, by induction, we have that the lemma holds for all $s, \sigma_1, \sigma_2, \sigma_1', \sigma_2'$. $\square$

To enable modularity, it must be sound to reason about a specification rather than an implementation. Therefore, it is imperative to show that whenever the *specified* effects of two operation calls are non-interfering, the *actual* effects of any valid bodies for those operations are also non-interfering.

Lemma 3.4.3. *Given two operation contracts $o_1 = (i_1, \pi_1, pre_1, post_1, \mathcal{S}_1)$ and $o_2 = (i_2, \pi_2, pre_2, post_2, \mathcal{S}_2)$, bodies b_1, b_2 such that $(b_1 \mapsto o_1 \wedge b_1 \rightarrowtail o_1) \wedge (b_2 \mapsto o_2 \wedge b_2 \rightarrowtail o_2)$, and state σ,*

$$\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma) \Rightarrow \mathcal{A}_\sigma(b_1) \ddagger \mathcal{A}_\sigma(b_2).$$

Proof. Let bodies b_1 and b_2 be in realizations with fields F_1, F_2 and interference correspondences $\mathcal{I}_1, \mathcal{I}_2$, respectively. Without loss of generality, we work only with b_1, o_1, and $\mathcal{I}_1$, for which we will omit subscripts; the proof and definitions are analogous for b_2, o_2, and $\mathcal{I}_2$.

By eq. (3.16), $\mathcal{A}_\sigma(b)$ is the application of function $\hat{\mathcal{I}}$ to the actual effect of the statements s that make up b. To obtain a useful instance of lemma 3.1.10, we first define the function j_σ as follows.

$$j_\sigma(x) = \begin{cases} \mathcal{I}(\sigma, x) & x \in \hat{\mathcal{P}}(F) \\ x & \text{otherwise} \end{cases} \tag{3.26}$$

From this, we observe that $\hat{\mathcal{I}}(\sigma, e) = \mathcal{R}\big(J_\sigma(e.A), J_\sigma(e.P)\big)$ where J_σ is defined relative to j_σ as in lemma 3.1.10, that is, element-wise application of j_σ to a set of objects.

We further observe that $\mathcal{A}_\sigma\big(o(p)\big) = \mathcal{S}(\sigma)[\pi \to p]$ for some sequence of arguments p; that is, by definition 3.2.12, $\hat{\mathcal{I}}\big(\sigma, \mathcal{A}_\sigma(s)\big) \sqsubseteq \mathcal{S}(\sigma)[\pi \to p]$.

Next, by lemma 3.1.8, if $\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma)$, then $\mathcal{A}_\sigma\big(o_1(p_1)\big) \ddagger \mathcal{A}_\sigma\big(o_2(p_2)\big)$. By lemma 3.1.10 (and lemmas 3.1.6 and 3.1.8), if $\mathcal{A}_\sigma\big(o_1(p_1)\big) \ddagger \mathcal{A}_\sigma\big(o_2(p_2)\big)$ then $\mathcal{A}_\sigma(b_1) \ddagger \mathcal{A}_\sigma(b_2)$. Therefore, $\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma) \Rightarrow \mathcal{A}_\sigma(b_1) \ddagger \mathcal{A}_\sigma(b_2)$. □

Although enabling modularity in reasoning is a primary concern, it remains important in parallel programs to have some knowledge of the implementation. Lemma 3.4.4 below ensures that reasoning about these implementation details does not break soundness. This lemma justifies the function inlining transformation that is a ubiquitous optimization of modern compilers.

Lemma 3.4.4. *An operation call can be replaced with any correct body without compromising correctness. That is,*

$$\forall s, t, Op, o, p, b : (s; o(p); t \mapsto Op \wedge b \mapsto o) \Rightarrow (s; b; t \mapsto Op)$$

(Renaming within the body to match argument names and to avoid name collisions is left implicit.)

Proof. By eq. (3.10),

$$s; o(p); t \mapsto Op \Rightarrow$$
$$\big(\forall \sigma, \sigma', \sigma'' : \sigma \xrightarrow{s} \sigma' : \sigma' \xrightarrow{o(p)} \sigma'' \Rightarrow \sigma'' \vdash \mathsf{pre}(t)\big).$$

By definition 3.2.11 and eq. (3.3),

$$b \mapsto o \Rightarrow$$
$$\big(\forall \sigma, \sigma' : \sigma \xrightarrow{b} \sigma' \Rightarrow \sigma \xrightarrow{o(p)} \sigma'\big) \Rightarrow$$
$$\forall \sigma, \sigma' : \big(\sigma \vdash o.pre \Rightarrow \sigma \vdash \mathsf{pre}(b)\big) \wedge \big(\sigma' \vdash \mathsf{post}(b) \Rightarrow \sigma' \vdash o.post\big).$$

Therefore,

$$s; o(p); t \mapsto Op \Rightarrow$$
$$\big(\forall \sigma, \sigma', \sigma'' : \sigma \xrightarrow{s} \sigma' : \sigma' \xrightarrow{b} \sigma'' \Rightarrow \sigma'' \vdash \mathsf{pre}(t)\big).$$

Then, by definition 3.2.11 and eq. (3.10),

$$\big(s; o(p); t \mapsto Op \wedge b \mapsto o\big) \Rightarrow \big(s; b; t \mapsto Op\big).$$

□

This contract/body replacement can go the other way, too, given a strong enough contract. Define $\hat{o}_{p,q}$ to be the operation contract induced by operation contracts p and q as follows (when p and q are understood, they are left out and we refer simply to $\hat{o}$).

$$\begin{pmatrix} \pi : p.\pi \circ q.\pi, \\ \mathit{pre} : \mathsf{pre}(p(p.\pi); q(q.\pi)) \vee \mathsf{pre}(q(q.\pi); p(p.\pi)), \\ \mathit{post} : \mathsf{post}(p(p.\pi); q(q.\pi)) \wedge \mathsf{post}(q(q.\pi); p(p.\pi)), \\ \mathcal{S}(\sigma) : p.\mathcal{S}(\sigma) \sqcup q.\mathcal{S}(\sigma) \end{pmatrix} \qquad (\hat{o}_{p,q})$$

Lemma 3.4.5.

$$\forall s, t_1, t_2, r, Op : s; t_1; t_2; r \mapsto Op \wedge : s; \hat{o}_{t_1,t_2}; r \mapsto Op.$$

Proof. Suppose that $s; t_1; t_2; r \mapsto Op$ for some operation contract $Op = (i, \pi, \mathit{pre}, \mathit{post}, \mathcal{S})$. Then, by definition 3.2.11 and eq. (3.3),

$$\forall \sigma, \sigma' : \sigma \xrightarrow{s;t_1;t_2;r} \sigma' \Rightarrow \sigma \vdash \mathit{pre} \wedge \sigma' \vdash \mathit{post}.$$

By eq. (3.10), the state of the program after s satisfies both $\mathsf{post}(s)$ and $\mathsf{pre}(t_1; t_2; r)$ and the state after $s; t_1; t_2$ satisfies both $\mathsf{post}(s; t_1; t_2)$ and $\mathsf{pre}(r)$. It is also true that

$$\mathsf{pre}(t_1; t_2; r) \Rightarrow \mathsf{pre}(t_1; t_2) \wedge \mathsf{post}(s; t_1; t_2) \Rightarrow \mathsf{post}(t_1; t_2).$$

By $(\hat{o}_{p,q})$,

$$\mathsf{pre}(\hat{o}_{t_1,t_2}) = (\mathsf{pre}(t_1;t_2) \vee \mathsf{pre}(t_2;t_1)) \wedge$$
$$\mathsf{post}(\hat{o}_{t_1,t_2}) = (\mathsf{post}(t_1;t_2) \wedge \mathsf{post}(t_2;t_1)).$$

Clearly, then, $\mathsf{pre}(t_1;t_2) \Rightarrow \mathsf{pre}(\hat{o}_{t_1,t_2})$ and $\mathsf{post}(\hat{o}_{t_1,t_2}) \Rightarrow \mathsf{post}(t_1;t_2)$. Therefore, whatever state the program is in after s (which we saw above must satisfy $\mathsf{pre}(t_1;t_2;r)$ and therefore $\mathsf{pre}(t_1;t_2)$) also satisfies $\mathsf{pre}(\hat{o}_{t_1,t_2})$. Further, whatever state the program is in after $s;\hat{o}_{t_1,t_2}$ (which must satisfy $\mathsf{post}(s;\hat{o}_{t_1,t_2})$ and therefore $\mathsf{post}(\hat{o}_{t_1,t_2})$) also satisfies $\mathsf{post}(\hat{o}_{t_1,t_2})$. Therefore,

$$\sigma \xrightarrow{t_1;t_2} \sigma' \Rightarrow \sigma \xrightarrow{\hat{o}_{t_1,t_2}} \sigma',$$

and therefore

$$s;t_1;t_2;r \mapsto \mathit{Op} \Rightarrow s;\hat{o}_{t_1,t_2};r \mapsto \mathit{Op}.$$

□

3.4.1 Commutability of Non-Interfering Statements

Theorem 4 is the main result of this chapter. Using this result, modular proof rules for the parallel composition of non-interfering statements can be formulated that generalize previous results. One application of this result (and of the A/P Calculus and the programming model above) is the system for reasoning about non-interference contracts introduced in chapter 5.

Theorem 4 (Non-interfering Statements Commute)**.** *For any two operations o_1, o_2 with valid bodies s_1, s_2, if*

$$\forall \sigma, p_1, p_2 :$$
$$\big(\sigma \vdash \mathsf{pre}(o_1(p_1);o_2(p_2)) \vee \mathsf{pre}(o_2(p_2);o_1(p_1)) : o_1.\mathcal{S}(\sigma) \ddagger o_2.\mathcal{S}(\sigma)\big),$$

then both of the following are true.

1. *$s_1; s_2$ is a valid operation body for $\hat{o}_{o_1,o_2}$ if and only if $s_2; s_1$ is a valid operation body for $\hat{o}_{o_1,o_2}$. That is,*

$$(s_1; s_2 \rightarrowtail \hat{o} \wedge s_1; s_2 \mapsto \hat{o}) \Leftrightarrow (s_2; s_1 \rightarrowtail \hat{o} \wedge s_2; s_1 \mapsto \hat{o})$$

2. `cobegin` *$s_1 \| s_2$* `end` *is a valid operation body for $\hat{o}_{o_1,o_2}$ if and only if $s_1; s_2$ is a valid operation body for $\hat{o}_{o_1,o_2}$. That is,*

$$(\texttt{cobegin } s_1 \| s_2 \texttt{ end} \rightarrowtail \hat{o} \wedge \texttt{cobegin } s_1 \| s_2 \texttt{ end} \mapsto \hat{o}) \Leftrightarrow$$
$$\big(s_1; s_2 \rightarrowtail \hat{o} \wedge s_1; s_2 \mapsto \hat{o}\big).$$

Proof. First, by eq. (3.21), definition 3.2.11, and theorems 1 and 2,

$$s_1; s_2 \rightarrowtail \hat{o} \wedge s_2; s_1 \rightarrowtail \hat{o} \ \wedge \texttt{cobegin } s_1 \| s_2 \texttt{ end} \rightarrowtail \hat{o},$$

i.e., the actual effect of the sequential composition of the two statements in either order is covered by the specified effect in $\hat{o}$, as is the actual effect of their parallel composition.

We show that the implements relation ($\mapsto$) holds by induction on the "abstraction level" of the statements (how far removed they are from primitive operations).

Base Case

The base case is when s_1 and s_2 both consist of a single call to `Prim` (and o_1, o_2 are equivalent to `Prim`). First, $\mathsf{pre}(s_1; s_1)$ and $\mathsf{pre}(s_2; s_2)$ are both true (*i.e.*, $\hat{o}_{o_1,o_2}.pre = \mathsf{true}$). By (`Prim`), each sequence of parameters p_1 and p_2 is divided into two subsequences, π_A and π_P. Additionally, the value of each object in π_P is the same after the operation as it was before the operation. Next, because `Prim` places *entire objects* into the A and P sets of the (constant) specified effect $\mathcal{S}$ (*i.e.*, the objects are not sub-divided based on their partitions), the two calls to `Prim` are noninterfering (*i.e.*, $\mathcal{A}_\sigma(s_1) \ddagger \mathcal{A}_\sigma(s_2)$) only when either no objects are shared between the two calls or all of the shared objects are in π_P for both calls. Therefore, by the Frame Rule (3.4), all shared objects have the same value after $s_1; s_2$ as they have after $s_2; s_1$—specifically, their value has not changed. Moreover, by lemma 3.4.1 and eq. (3.4), any object not an argument in either s_1 or s_2 has the same value before and after $s_1; s_2$ and $s_2; s_1$. Finally, we consider those objects that appear in π_A for either s_1 or s_2 (but not both)—without loss of generality consider those objects that appear in s_1 but not s_2. Each object x that appears in s_1 but not s_2 has, after s_1 is complete, value x'. But x does not appear in s_2, so its value after s_2 is *also* x', so the value of x after $s_1; s_2$ and $s_2; s_1$ is the same: x'. Therefore, if $\mathcal{A}_\sigma(s_1) \ddagger \mathcal{A}_\sigma(s_2)$, $\mathsf{post}(s_1; s_2) = \mathsf{post}(s_2; s_1)$; therefore by eq. (3.10) $s_1; s_2 \mapsto \hat{o} \Leftrightarrow s_2; s_1 \mapsto \hat{o}$.

Since `Prim` is atomic, the parallel execution of two calls to `Prim` is exactly equivalent to one of the sequential orderings of those calls (which themselves are equivalent, as shown above); therefore, $s_1; s_2 \mapsto \hat{o} \Leftrightarrow \mathtt{cobegin}\ s_1 \| s_2\ \mathtt{end} \mapsto \hat{o}$.

Inductive Step

Assume the theorem holds for each consitituent statement in s_1 and s_2 which are valid bodies for some operations o_1, o_2 with preconditions pre_1, pre_2, postconditions $post_1, post_2$, and specified effects $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. Further assume σ is such that $\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma)$ and $\sigma \vdash pre_1 \vee pre_2$.

We adopt the following conventions and notations for the remainder of the proof:

- Every statement is an operation call. This does not reduce generality because any other kind of statement (such as a conditional statement or loop) can be refactored as an operation contract and associated body.
- An operation body s is treated as the sequential composition of statements $s[1]; s[2]; \ldots; s[|s|]$.
- $s[i, j)$ is the subsequence of statements in s from the ith through $(j-1)$th statements, well-defined whenever $1 \leq i \leq j \leq |s| + 1$.
- $b_{i[j]}$ is a valid operation body for the operation call in statement $s_i[j]$.

By eq. (3.21), lemmas 3.1.8 and 3.1.10, and theorems 1 and 2,

$$\forall \sigma, i, j : \sigma \xrightarrow{s_1[1,i)} \sigma_i \wedge \sigma \xrightarrow{s_2[1,j)} \sigma_j : \mathcal{A}_{\sigma_i}(s_1[i]) \ddagger \mathcal{A}_{\sigma_j}(s_2[j]). \tag{3.27}$$

Without loss of generality, we consider $s_1; s_2$ (an analogous proof applies to $s_2; s_1$). The statements $s_1[|s_1|]$ and $s_2[1]$ induce $\hat{o}_s$ according to eq. ($\hat{o}_{p,q}$). Let σ' be such that $\sigma \xrightarrow{s_1[1,|s_1|)} \sigma'$. By lemma 3.4.4, for all Op,

$$s_1; s_2 \mapsto Op \Rightarrow b_{1[1]}; b_{1[2]}; \ldots; b_{1[|s_1|]}; b_{2[1]}; b_{2[2]}; \ldots; b_{2[|s_2|]} \mapsto Op$$

By eq. (3.27),

$$\mathcal{A}_{\sigma'}(s_1[|s_1|]) \ddagger \mathcal{A}_{\sigma'}(s_2[1]).$$

So, by inductive hypothesis,

$$b_{1[|s_1|]}; b_{2[1]} \mapsto \hat{o}_s \Leftrightarrow b_{2[1]}; b_{1[|s_1|]} \mapsto \hat{o}_s.$$

By eqs. (3.9) and (3.10) and lemma 3.4.5,

$$\begin{aligned} s_1; s_2 \mapsto Op &\Rightarrow s_1[1, |s_1|); s_1[|s_1|]; s_2[1]; s_2[1, |s_2| + 1) \mapsto Op \\ &\Rightarrow s_1[1, |s_1|); \hat{o}_s; s_2[1, |s_2| + 1) \mapsto Op. \end{aligned}$$

By lemma 3.4.4,

$$b_{1[1]}; b_{1[2]}; \ldots ; b_{1[|s_1|-1]}; b_{2[1]}; b_{1[|s_1|]}; b_{2[2]}; b_{2[3]}; \ldots ; b_{2[|s_2]} \mapsto Op.$$

Now, the statements $s_1[|s_1| - 1]$ and $s_2[1]$ induce $\hat{o}'_s$; by the same reasoning as above we have that

$$b_{1[1]}; b_{1[2]}; \ldots ; b_{1[|s_1|-2]}; b_{2[1]}; b_{1[|s_1|-1]}; b_{1[|s_1|]}; b_{2[2]}; b_{2[3]}; \ldots ; b_{2[|s_2]} \mapsto Op,$$

and we can continue applying the same reasoning to sift $b_{2[1]}$ to the beginning of the sequence of statements:

$$b_{2[1]}; b_{1[1]}; b_{1[2]}; \ldots ; b_{1[|s_1|]}; b_{2[2]}; b_{2[3]}; \ldots ; b_{2[|s_2|]} \mapsto Op.$$

Furthermore, we can sift each statement $b_{2[i]}$ to the front of the $b_{1[j]}$'s, and we have that

$$b_{2[1]}; b_{2[2]}; \ldots ; b_{2[|s_2|]}; b_{1[1]}; b_{1[2]}; \ldots ; b_{1[|s_1|]} \mapsto Op$$

Since $b_{1[1]}; b_{1[2]}; \ldots ; b_{1[|s_1|]} \mapsto s_1$ and $b_{2[1]}; b_{2[2]}; \ldots ; b_{2[|s_2|]} \mapsto s_2$, it follows that

$$\sigma \xrightarrow{b_{2[1]};b_{2[2]};\ldots;b_{2[|s_2|]};b_{1[1]};b_{1[2]};\ldots;b_{1[|s_1|]}} \sigma' \Rightarrow \sigma \xrightarrow{s_2;s_1} \sigma'.$$

Therefore, by definition 3.2.11,

$$b_{2[1]}; b_{2[2]}; \ldots ; b_{2[|s_2|]}; b_{1[1]}; b_{1[2]}; \ldots ; b_{1[|s_1|]} \mapsto Op \Rightarrow s_2; s_1 \mapsto Op.$$

By transitivity of $\Rightarrow$, then, we have that

$$s_1; s_2 \mapsto Op \Rightarrow s_2; s_1 \mapsto Op$$

Therefore,

$$s_1; s_2 \mapsto \hat{o}_{o_1,o_2} \Rightarrow s_2; s_1 \mapsto \hat{o}_{o_1,o_2}.$$

By swapping s_1 and s_2 and performing the same sifting process as above, we conclude

$$s_2; s_1 \mapsto \hat{o}_{o_1,o_2} \Rightarrow s_1; s_2 \mapsto \hat{o}_{o_1,o_2}.$$

Therefore,

$$s_1; s_2 \mapsto \hat{o} \Leftrightarrow s_2; s_1 \mapsto \hat{o}.$$

Through the sifting process, it was established that every possible interleaving of the consituent statements in s_1 and s_2 is *also* a valid operation body for $\hat{o}_{o_1,o_2}$. Therefore, if $s_1; s_2$ is a valid operation body for $\hat{o}_{o_1,o_2}$, then `cobegin` $s_1 \| s_2$ `end` is also a valid operation body for $\hat{o}_{o_1,o_2}$. □

3.5 Applying Theorem 4

The A/P Calculus and programming model have implications for writing and reasoning about parallel programs. In particular, theorem 4 implies that non-interfering operation calls commute *even when their specifications indicate otherwise.* Consider the operation contracts (`Rem`) and (`Add`) that operate on a Queue (with entries of type T) with type specification in eq. (3.28). Each object of type $\mathtt{Queue}_T$ has a partition with two pieces named h and t, *i.e.*, $\forall q : [q : \mathtt{Queue}_T] : \mathcal{P}(q) = \{q@h, q@t\}$ (a specific piece of a partition of an object is denoted with @, *e.g.*, $q@h$ refers to the element of q's partition named h).

$$\mathtt{Queue}_T = \big(T^*, \langle\rangle, \{\mathit{enqueue}, \mathit{dequeue}, \mathit{isEmpty}\}\big) \tag{3.28}$$

An object of type $\mathtt{Queue}_T$ is a modeled as a string with entries of type T.

$\mathtt{Queue}_T.v_0$ is the empty string.

$\mathtt{Queue}_T$ provides these operations (their contracts are as would be expected).

$$\begin{pmatrix} i: \texttt{RemoveAnEnd}, \\ \pi: \langle q, x \rangle, \\ pre: [q: \texttt{Queue}_T] \land [x: T] \land |q| > 0, \\ post: q_0 = q \circ \langle x \rangle \lor q_0 = \langle x \rangle \circ q, \\ \mathcal{S}(\sigma): (\{q@h\} \cup \mathcal{P}(x), \emptyset) \end{pmatrix} \tag{Rem}$$

$$\begin{pmatrix} i: \texttt{AddAnEnd}, \\ \pi: \langle q, y \rangle, \\ pre: [q: \texttt{Queue}_T] \land [y: T], \\ post: q = q_0 \circ \langle y_0 \rangle \lor q = \langle y_0 \rangle \circ q_0, \\ \mathcal{S}(\sigma): (\{q@t\} \cup \mathcal{P}(y), \emptyset) \end{pmatrix} \tag{Add}$$

Next, consider the program fragment below in which objects q, u, and v are initialized previously and have some values.

Listing 3.4: Program fragment making operation calls on a Queue.

```
AddAnEnd(q, v);
RemoveAnEnd(q, u);
```

Assume the program is in the following state at the beginning of the fragment:

$$q = \langle 10, 20, 30 \rangle \land u = 4$$

By looking solely at the behavioral specifications of the two operations (*i.e.*, the *pre* and *post* in the contracts), it can be shown (via eq. (3.10)) that the program is in one of the following four states after the two statements:

- $q = \langle 4, 10, 20 \rangle \land v = 30$

- $q = \langle 20, 30, 4 \rangle \wedge v = 10$
- $q = \langle 4, 20, 30 \rangle \wedge v = 10$
- $q = \langle 10, 20, 30 \rangle \wedge v = 4$

However, the possible states can be whittled down by recognizing that the two operation calls are *non-interfering* based on their specified effects. Using theorem 4, we recognize that the program above (more precisely, the sequential composition of two valid operation bodies for `RemoveAnEnd` and `AddAnEnd`) is a valid operation body for the following contract, (`Rot`).

$$\begin{pmatrix} i : \texttt{RotateOnce}, \\ \pi : \langle q, u, v \rangle, \\ \mathit{pre} : [q : \texttt{Queue}_T] \wedge [u : T] \wedge [v : T], \\ \mathit{post} : \begin{pmatrix} \big(q = \langle u_0 \rangle \circ q_0[0, |q| - 1) \wedge \langle v \rangle = q_0[|q| - 1, |q|)\big) \vee \\ \big(q = q_0[1, |q|) \circ u_0 \wedge \langle v \rangle = q_0[0, 1)\big) \end{pmatrix}, \\ \mathcal{S}(\sigma) : \big(\mathcal{P}(q) \cup \mathcal{P}(u) \cup \mathcal{P}(v), \emptyset\big) \end{pmatrix} \tag{Rot}$$

Therefore, we can conclude that the state at the end of the program in listing 3.4 is one of the following two states (rather than one of the four above):

- $q = \langle 4, 10, 20 \rangle \wedge v = 30$
- $q = \langle 20, 30, 4 \rangle \wedge v = 10$

In fact, the theorem leads to even a stronger conclusion. The two states above are also the only possible states after either of the following two program fragments given the same initial conditions:

```
AddAnEnd(q, v);
RemoveAnEnd(q, u);
```

```
cobegin
  RemoveAnEnd(q, v);
  AddAnEnd(q, u);
end
```

Moreover, observe that except for type restrictions, $\mathtt{Rot}.pre \equiv \mathtt{true}$. Therefore, even when $|q| = 0$, the parallel execution of `AddAnEnd` and `RemoveAnEnd` can safely proceed even though the precondition for `RemoveAnEnd` would suggest otherwise.

3.6 Comparison to Region Logic

The actual effect of a statement maps approximately to the effects summaries in region-logic based programming languages such as DPJ [40] except that the actual effect depends on the behavioral semantics of the program. To concretize this mapping, it is useful to define the effect of a statement on a conservative basis without regard for the initial state. For this purpose we define the *syntactic effect*, denoted $\mathcal{A}(s)$, so named because it is syntactically derivable from the program text. Formally,

$$\mathcal{A}(s) = \bigsqcup \sigma : \mathcal{A}_\sigma(s).$$

Because the syntactic effect is always more conservative than the actual effect, it can be used in place of the actual effect without compromising soundness (although there is a penalty to completeness). That is,

$$\forall s, \sigma : \mathcal{A}_\sigma(s) \sqsubseteq \mathcal{A}(s).$$

In the effects summary of a DPJ method, annotations identify which regions of memory are written to or read from by the body. Those regions correspond to *pieces* in our object model. In the effect, the A set corresponds to the written-to regions and the P set the read-from regions. However, each field in a DPJ implementation is placed into a single region; this is in contrast with the partition of an object in this chapter, whereby an object might be split into several pieces—perhaps even dynamically based on the (abstract) values of the fields. In this way, the A/P Calculus generalizes other results by increasing abstraction and modularity.

3.7 Conclusions

The calculus introduced in section 3.1 provides a general foundation for reasoning about the non-interference of parallel programs. It was used in the definition of a model of programming in which non-interfering statements commute even when their specifications indicate otherwise, and a simple program was reasoned about in this model. Chapters 4 and 5 build upon these foundations and show how more complex parallel programs can be reasoned about and proven correct.

Chapter 4: Array Abstractions to Amortize the Reasoning Cost of Parallel Client Code

Traditional parallel implementations of many algorithms (*e.g.*, divide-and-conquer, map-reduce, and producer-consumer) are often subtly difficult for automated verification systems to reason about because the verifier must ensure that the parallel calls do not write to the same memory location at the same time. Moreover, small modifications to these programs (*e.g.*, dividing a data structure into four parts instead of two) can result in having to re-prove the program from scratch, even though the new proof obligations are mostly identical to the previous ones. Finally, aliased references cause problems because it is in general impossible to decide via local reasoning whether two references are aliases to one another.

This chapter focuses on the `SplittableArray` abstraction that permits a client to split an array into sub-arrays of contiguous indices and ensures by construction that all sub-arrays of the array that may be accessed are separate from one another and which leverages clean semantics to ensure that no dangerous data sharing can occur. This array abstraction simplifies proofs of non-interference between parallel threads, often turning them into simple syntactic checks that do not require reasoning about the values of objects. The abstraction can be implemented using a traditional array that is shared among many instances in order to maintain desirable performance characteristics of such data structures: constant-time split, combine, and lookup.

4.1 A Motivating Example

Listing 4.1 shows a generic recursive, parallel divide-and-conquer method written in a Java-like language that mutates each entry in the array `arr`. The parallel calls to `mutateEach` share two arguments: `arr` and `mid`. This data-sharing between concurrent threads of execution is *safe* (*i.e.*, non-interfering) only when `arr` is written by different threads in compatible (non-overlapping) ways and `mid` (in this case, a primitive variable) is copied in each call. If either of these criteria are not met (*e.g.*, there is aliasing within `arr`), then no guarantees of non-interference can be made and reasoning is complicated substantially.

Listing 4.1: A generic recursive, parallel divide-and-conquer solution using a Java-like language.

```
void mutateEach(T[] arr, int lowEnough, int tooHigh) {
    if (tooHigh - lowEnough > 1) {
        int mid = (lowEnough + tooHigh) / 2;
        cobegin {
            mutateEach(arr, lowEnough, mid);
            mutateEach(arr, mid, tooHigh);
        }
    } else if (tooHigh > lowEnough) {
        mutate(arr[lowEnough]);
    }
}
```

4.1.1 Verification Challenges with this Approach

The approach in listing 4.1 presents several challenges to verification, and especially to modular verification. These challenges are addressed in order of increasing complexity of reasoning: first the case is considered where `arr` is an array of primitives or immutable objects, then when `arr` is an array of mutable objects, and finally when `arr` is an array of objects using a shared representation.

The Overlapping Array Intervals Problem

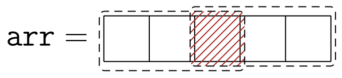


Figure 4.1: `mutateEach` might affect the closed interval $[lowEnough, tooHigh]$ or the half-open interval $[lowEnough, tooHigh)$.

The simplest case is when `T` is a primitive or immutable type. To verify the functional correctness of this method body relative to a formal version of its specification, it is helpful to show that the parallel portion of the code does not introduce any nondeterminism[5] through data races. For this, it is sufficient to show that the two recursive calls are non-interfering. Showing non-interference, especially when each element of the array might be modified, requires showing that the intervals $[\mathit{lowEnough}, \mathit{mid})$ and $[\mathit{mid}, \mathit{tooHigh})$ are disjoint. This is not a hard problem in this simple case, but suppose (*e.g.*, for performance reasons) that a programmer modified this program to split `A` into 4 parts: $[\mathit{lowEnough}, q_1)$, $[q_1, \mathit{mid})$, $[\mathit{mid}, q_3)$, and $[q_3, \mathit{tooHigh})$. Now there are four intervals which must be shown to be pairwise disjoint. In the general case, showing mutual disjointness of n sets of indices is non-trivial, and it is certainly

[5]It is possible for a program to exhibit nondeterminism and still be correct, but for now we are concerned only with provably deterministic parallel programs.

not readily scalable (the number of pairs increases quadratically with n). The explicit split/combine operations in the `SplittableArray` abstraction discussed in section 4.3 are motivated by this problem.

This problem is exacerbated when the partition is not into contiguous segments of the array, for example a partition of `arr` into the entries with even indices and those with odd indices. A desire for enabling the use of arbitray index sets motivates a more general array abstraction, `IndexPartitionableArray`, discussed and specified in section 4.4.2.

Challenges Related to Aliasing

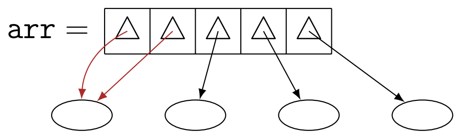


Figure 4.2: An array with aliases between its elements.

Next we identify challenges posed when `T` is a mutable reference type such as `Stack`. When reasoning about the code in `mutateEach`, a requirement for the non-interference of the parallel section of code is the total independence of each stack in `arr`. In particular, a specification of this method written as a Hoare triple in Separation Logic might look similar to the specification in eq. (4.1).

$$\left\{\bigodot_{i=l}^{h-1} \mathsf{list}\alpha_i\left(\mathtt{arr}_0[i], \mathsf{nil}\right)\right\} \mathtt{mutateEach}(\mathtt{arr}, \mathtt{l}, \mathtt{h}); \left\{\bigodot_{i=l}^{h-1} \mathsf{list}\alpha'_i\left(\mathtt{arr}_0[i], \mathsf{nil}\right)\right\} \tag{4.1}$$

In English, the meaning of this specification is as follows. The precondition, $\left\{\bigodot_{i=l}^{h-1} \mathsf{list}\alpha_i\,(\mathtt{arr}_0[i], \mathsf{nil})\right\}$, states that for each $i \in [l, h)$, the initial value of the i-th element of array `arr` is a nil-terminated (singly linked) list with abstract value α_i, and that each of these lists is separate in the heap (that is, they share no nodes). The postcondition states that the abstract value of each element of `arr` has changed to α'_i (and, again, that the lists are separate).

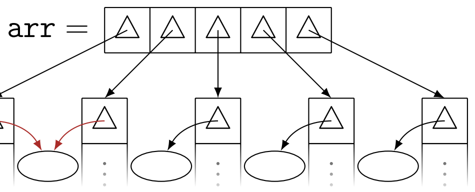


Figure 4.3: Even if the precondition of eq. (4.1) is satisfied, there may be aliases *within* the elements of `arr`.

A major problem with the specification in eq. (4.1) is that it does not preclude aliasing within the stacks in `arr`. For example, if the top element of `arr[0]` is an alias to the top element of `arr[1]`, the picture might look like fig. 4.3. Note that it is still true that for some α, β, $\{\mathsf{list}\ \alpha\,(\mathtt{arr}[0], \mathsf{nil}) * \mathsf{list}\ \beta\,(\mathtt{arr}[1], \mathsf{nil})\}$, so the precondition is satisfied. The concern with this scenario is that if the mutation performed by the `mutate` operation is not idempotent, the result will not be correct (in fact, if the top of one stack is an alias to an element of a stack that is not the top, even an idempotent mutation will result in incorrect behavior). In a language with clean semantics, we can rely on the fact that separate variables act as separate objects to guarantee there is no dangerous aliasing.[6]

[6]While a modified specification could be written in separation logic to handle each particular situation, it is extremely challenging to generalize the specification [136].

The Shared Representation Problem

A more subtle issue arises when instances of `T` use a shared representation. For example, the precondition as written would not be satisfied if the stack implementation were swapped out for one based on a shared "cactus stack", as in fig. 4.4, even though such an implementation could provide correct behavior. In the figure, the separation logic specification breaks down because it is *not* the case that $\mathsf{list}\ \alpha(\mathtt{arr}[0], \mathsf{nil}) * \mathsf{list}\ \beta(\mathtt{arr}[1], \mathsf{nil})$ for any α, β because they share a node. This lack of modularity demonstrates a need for abstraction in the specification of concurrent programs to facilitate reusable code that can remain proven to be correct even when different underlying data structures are used.

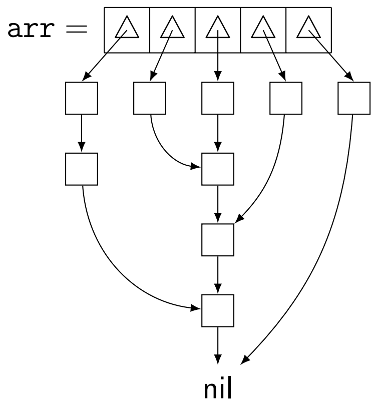


Figure 4.4: If the stacks in `arr` use a cactus stack realization, the separation logic specification breaks down.

4.2 The RESOLVE Programming Language

The broad context for this dissertation is RESOLVE [129, 110, 62, 45, 126], an imperative, object-based programming language that also includes a specification framework to support the automated, modular verification of software. In RESOLVE, a *realization* provides executable code for a *concept*, which contains behavioral specifications for several operations in terms of abstract state. The operation contracts that make up the behavioral specification are composed of pre- and post-conditions that utilize mathematical notation appropriate for the abstract state. Most routine aliasing is avoided in RESOLVE by using swap as the fundamental data movement operation [73].

In addition to pre- and post-condition specifications, operation contracts include "parameter modes", whereby the modification frame is defined and certain behaviors are summarized. There are five parameter modes:

- **Restores**: The value of a *restores*-mode parameter is the same at the end of the operation as it was at the beginning.
- **Clears**: The value of a *clears*-mode parameter is an initial value for its type at the end of the operation (*e.g.*, a *clears*-mode `Integer` parameter would have the value 0 at the end of the operation, or a set would be made empty).
- **Updates**: The value of an *updates*-mode parameter might be changed by the operation, and its initial value impacts the behavior of that operation.

- **Replaces**: The value of a *replaces*-mode parameter might be changed by the operation, and its initial value does not have an impact on the behavior of that operation.

- **Preserves**: The value of a *preserves*-mode parameter is not changed—even temporarily—by the operation.[7]

4.2.1 Clean and Rich Semantics

One way to address the challenges discussed above is by eliminateing aliases entirely. RESOLVE does this through a mechanism called *clean semantics*, by which separate variables are reasoned about as separate entities. Reasoning with clean semantics helps to keep reasoning devoid of complicated frame and heap assertions. Rich semantics is a mechanism by which variables are reasoned about only in terms of values in arbitrary mathematical domains and never in terms of memory locations. Reasoning with rich semantics ensures that a client of a software component need not concern themselves with implementation details. The soundness of rich semantics is guaranteed by clean semantics. Rich semantics and a value-based reasoning framework are necessary for the implementation of conditional effects in the non-interference contracts introduced in chapter 5.

[7]Preserves is necessary because in the realm of concurrency, restores-mode is not sufficient to ensure that two concurrent operations that share a restores-mode argument do not write to the same memory location since it does not preclude the temporary modification of a parameter during the execution of an operation.

Notation

The syntax used in the listings in this chapter and the next is based on RESOLVE except that traditional mathematical notation is used for the specifications to improve readability. The language has textual equivalents that would appear in a real program.

4.3 The SplittableArray Abstraction

The `SplittableArray` abstraction is a novel array abstraction that amortizes the cost of reasoning about parallel programs such as the one presented in listing 4.1. It provides the client with operations that divide the array at some split point into two sub-arrays with contiguous indices. By virtue of RESOLVE's clean semantics the resulting sub-arrays may be reasoned about as totally independent objects. The behavioral specification of this component is shown in listing 4.2.

Listing 4.2: Abstract specification for `SplittableArray`.

```
concept SplittableArrayTemplate(type Entry)

  var Ids: ℘(ℤ)
    initialization ensures $Ids = \emptyset$

  type family SplittableArray is modeled by (
    Id: ℤ
    LowerBound: ℤ,
    UpperBound: ℤ,
    Contents: ℤ → Entry,
    SplitPoint: ℤ,
    PartsInUse: 𝔹
  )
  exemplar A
    constraint
      $A.LowerBound \leq A.UpperBound \ \wedge$
      $A.LowerBound \leq A.SplitPoint \leq A.UpperBound \ \wedge$
      $A.Id \in Ids$
    initialization ensures
      $A.LowerBound = 0 \wedge A.UpperBound = 0 \ \wedge$
```

```
        $\neg A.PartsInUse \wedge A.Id \notin \#Ids$
end

operation SetBounds(
        restores LB: Integer,
        restores UB: Integer,
        updates A: SplittableArray)
    requires $LB \leq UB \wedge \neg A.PartsInUse$
    ensures $A.LowerBound = LB \wedge A.UpperBound = UB \;\wedge$
        $\neg A.PartsInUse \wedge A.Id \notin \#Ids \wedge \#A.Id \notin Ids$

operation SetSplitPoint(
        restores i: Integer,
        updates A: SplittableArray)
    requires $A.LowerBound \leq i \wedge i \leq A.UpperBound \;\wedge$
        $\neg A.PartsInUse \;\wedge$
    ensures $A = \#A$ except $A.SplitPoint = i$

operation SwapEntryAt(
        restores i: Integer,
        updates A: SplittableArray,
        updates E: Entry)
    requires $\neg A.PartsInUse \;\wedge$
        $A.LowerBound \leq i < A.UpperBound$
    ensures $E = \#A.Contents(i) \;\wedge$
        $A = \#A$ except $A.Contents(i) = \#E$

operation Split(
        updates A: SplittableArray,
        replaces L: SplittableArray,
        replaces U: SplittableArray)
    requires $\neg A.PartsInUse$
    ensures $(A = \#A$ except $A.PartsInUse) \;\wedge$
        $L.InclLowerBound = A.InclLowerBound \;\wedge$
        $L.ExclUpperBound = A.SplitPoint \;\wedge$
        $U.LowerBound = A.SplitPoint \;\wedge$
        $U.UpperBound = A.ExclUpperBound \;\wedge$
        $L.Id = A.Id \wedge L.Contents = A.Contents \;\wedge$
        $U.Id = A.Id \wedge U.Contents = A.Contents \;\wedge$
        $\neg L.PartsInUse \wedge \neg U.PartsInUse$

operation Combine(
        updates A: SplittableArray,
        clears L: SplittableArray,
        clears U: SplittableArray)
    requires $A.PartsInUse\ \wedge$
        $\neg L.PartsInUse \wedge \neg U.PartsInUse\ \wedge$
        $L.InclLowerBound = A.InclLowerBound\ \wedge$
        $L.ExclUpperBound = A.SplitPoint\ \wedge$
        $U.InclLowerBound = A.SplitPoint\ \wedge$
        $U.ExclUpperBound = A.ExclUpperBound\ \wedge$
        $L.Id = A.Id \wedge U.Id = A.Id$
    ensures $A = \#A$ except $\big(\neg A.PartsInUse\ \wedge$
        $\forall (i : \mathbb{Z})($
          $(i < A.SplitPoint) \Rightarrow (A.Contents(i) = \#L.Contents(i))\ \wedge$
          $(i \geq A.SplitPoint) \Rightarrow (A.Contents(i) = \#U.Contents(i)))\big)$

operation LowerBound(preserves A: SplittableArray):
 Integer
    ensures $LowerBound = A.LowerBound$

operation UpperBound(preserves A: SplittableArray):
 Integer
    ensures $UpperBound = A.UpperBound$

operation SplitPoint(preserves A: SplittableArray):
 Integer
    ensures $SplitPoint = A.SplitPoint$

operation PartsInUse(preserves A: SplittableArray):
 Boolean
    ensures $PartsInUse = A.PartsInUse$

operation IdsMatch(
        preserves A1: SplittableArray,
        preserves A2: SplittableArray): Boolean
    ensures $IdsMatch = (A1.Id = A2.Id)$

end SplittableArrayTemplate
```

The math model of a `SplittableArray` is a tuple with several components. A client should view a `SplittableArray` as a function from integer indices to values of type `Entry`, addressable in the range $[LowerBound, UpperBound)$. The operations allow the client to set a split point within the addressable range, and to split the array into two subarrays at index `A.SplitPoint` (and to combine them back into the original array). The `PartsInUse` flag indicates whether the array has been split into subparts, and controls access to those parts.

Figure 4.5 visualizes how a `SplittableArray` can be partitioned. After a call to `Split`, a `SplittableArray` is inaccessable until its two parts are fused back together with a call to `Combine`. The pre- and post-conditions of those operations ensure the contiguity of the addressable ranges of the sub-arrays.

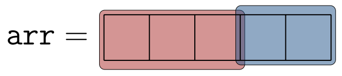


Figure 4.5: A `SplittableArray` can be partitioned into two sub-arrays of contiguous indices.

4.3.1 Divide-and-Conquer Client Code Using SplittableArray

A natural application of `SplittableArray` is in a parallel divide-and-conquer algorithm such as the `mutateEach` operation from listing 4.1. Using the component in that context dramatically simplifies the reasoning involved in formally verifying the correctness of such code. Listing 4.3 shows how `MutateEach` might be implemented and specified using `SplittableArray`.

Listing 4.3: A recursive, parallel divide-and-conquer solution using `SplittableArray`.

```
uses SplittableArray
uses Entry

operation MutateEach(updates A: SplittableArray);
```

```
  requires
    $A.LowerBound < A.UpperBound\ \wedge$
    $\neg A.PartsInUse$;
  ensures
    $\forall(i)(A.LowerBound \le i < A.UpperBound)($
      $A.Contents(i) = f(\#A.Contents(i)))$;
recursive procedure
  decreasing $A.UpperBound - A.LowerBound$;

  if (UpperBound(A) - LowerBound(A) > 1) then
    var A1, A2: SplittableArray
    var mid: Integer := (LowerBound(A) + UpperBound(A)) / 2
    SetSplitPoint(mid, A)
    Split(A, A1, A2)
    cobegin
      MutateEach(A1)
      MutateEach(A2)
    end
    Combine(A, A1, A2)
  else
    var entry: Entry
    var index: Integer := LowerBound(A)
    SwapEntryAt(A, index, entry)
    Mutate(entry)
    SwapEntryAt(A, index, entry)
  end
end MutateEach
```

As discussed in section 4.1, keeping verification of this code relatively simple involves showing that the operations inside the **cobegin** block are *non-interfering* as defined in chapter 3. Because the operation modifies `A`, if there were a shared array parameter between the two recursive calls, it would be challenging to show non-interference. Fortunately, however, the two calls to `MutateTops` inside the **cobegin** statement operate on different variables, so they are necessarily independent because of RESOLVE's clean semantics.

4.3.2 Reusability and Modifiability

The code in listing 4.3 is highly reusable. A client can use *any* type as the array's entry type. For example, if there is a cactus realization of a `Stack`, the client may use it without having to re-prove `MutateEach` or write a new specification.

It is also robust to modifications, requiring only simple proofs to be discharged in most cases. Consider an alternate approach to `MutateEach` where the array is split into four parts instead of two. Now the parallel section of the code might look like listing 4.4. Thanks to clean semantics, it is still a simple syntactic check to show that the four parallel calls are non-interfering. The one-time proof of disjointness in the intervals falls on the implementer of the `SplittableArray` specification—but is trivial unless the implementer opts for a shared realization such as one discussed in section 4.3.3.

Listing 4.4: The parallel section of a divide-and-conquer solution which splits `A` into four parts via consecutive calls to `Split`.

```
cobegin
  MutateEach(A1)
  MutateEach(A2)
  MutateEach(A3)
  MutateEach(A4)
end
```

4.3.3 Efficiently Realizing SplittableArray

A combination of clean semantics, careful component design, and robust specification capabilities has reduced the potentially complicated reasoning problem of showing non-interference in listing 4.1 to a purely syntactic check in listing 4.3, demonstrating a clear advantage of this approach over the traditional one. Importantly, these reasoning advantages can be achieved *without* compromising performance.

Although clean semantics permits a client to reason about the two sub-arrays `A1` and `A2` in listing 4.3 as if they were totally separate arrays, an efficient implementation of this concept would not make any copies of the array. The interface for the component was designed with a shared implementation in mind so that a realization could employ an underlying (traditional) array that is shared among all `SplittableArray` instances with the same `Id`. This design choice manifests itself in the use of the `Split` and `Combine` operations as pseudo-synchronization points by flipping `PartsInUse` and of preconditions to prevent access to the array while it is split. These choices ensure that at any time, there is only one array with each `Id` that can access any given index. Enabling such a shared implementation is important for preserving the performance benefits that programmers expect from parallel software: the operations `Split`, `Combine`, and `SwapEntryAt` can all be done in constant time.

4.4 Related Array Abstractions

The `SplittableArray` abstraction presented above is one member of a hierarchy of concurrency-ready array abstractions that can be used in multiple contexts [140]. The most general abstraction in this family, `IndexPartitionableArray`, may be partitioned on arbitrary indices rather than contiguous portions of the array. A third abstraction, `DistinguishedIndexArray`, allows a client to isolate one entry and operate on it separately from the rest of the array.

4.4.1 Distinguished Index Array

The `DistinguishedIndexArray` keeps a "distinguished index" whose corresponding entry is independent from the rest of the array. The

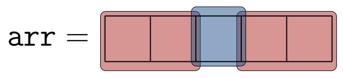


Figure 4.6: A `DistinguishedIndexArray` can be partitioned into two sub-arrays: one holding a single element and the other holding the rest of the array.

`DistinguishedIndexArray` is useful to simplify, *e.g.*, the verification of concurrency properties of an implementation of a bounded queue component (see chapter 5, section 5.6.1) or for an implementation of an array mapping algorithm. It could also be used as an underlying type in the implementation of a parallel iteration construct. The specification of `DistinguishedIndexArray` is in listing 4.5.

Listing 4.5: Specification for `DistinguishedIndexArray`.

```
concept DistinguishedIndexArrayTemplate(type Entry)

  var Ids: ℘(ℤ)

  type family DistinguishedIndexArray is modeled by (
    Id: ℤ,
    LowerBound: ℤ,
    UpperBound: ℤ,
    Contents: ℤ → Entry,
    DistinguishedIndex: ℤ,
    Domain: ℘(ℤ),
    RestInUse: 𝔹
  )
  exemplar A
    constraint
      $A.LowerBound \leq A.UpperBound \ \wedge$
      $A.Id \in Ids \wedge A.DistinguishedIndex \in A.Domain$
    initialization ensures
      $A.LowerBound = 0 \wedge A.UpperBound = 0 \wedge A.Id \notin \#Ids \ \wedge$
      $\neg A.RestInUse \wedge A.Domain = \{A.DistinguishedIndex\}$
  end

  operation SetBounds(
      restores LB: Integer,
      restores UB: Integer,
      updates A: DistinguishedIndexArray)
    requires $LB \leq UB \wedge \neg A.RestInUse$
    ensures $A.LowerBound = LB \wedge A.UpperBound = UB \wedge$
      $\neg A.RestInUse \wedge A.Id \notin \#Ids \wedge A.Id \in Ids \wedge A.Id \neq \#A.Id \wedge$
      $A.Domain = \{(i:\mathbb{Z})(A.LowerBound \leq i \wedge i < A.UpperBound)(i)\}$

  operation SwapDistinguishedEntry(
      updates A: DistinguishedIndexArray,
      updates E: Entry)
    requires $A.LowerBound \leq A.DistinguishedIndex \wedge$
      $A.DistinguishedIndex < A.UpperBound$
    ensures $E = \#A(A.DistinguishedIndex) \wedge$
      $A = \#A$ except $A(A.DistinguishedIndex) = \#E$

  operation RetrieveRest(
      updates A: DistinguishedIndexArray,
      replaces B: DistinguishedIndexArray)
    requires $\neg A.RestInUse$
    ensures ($A = \#A$ except $A.RestInUse$) $\wedge$
      $B.Domain =$
        $A.Domain \setminus \{A.DistinguishedIndex\} \cup B.DistinguishedIndex \wedge$
      $A.DistinguishedIndex \neq B.DistinguishedIndex \wedge$
      $EqExceptOn(A.Contents, B.Contents, A.DistinguishedIndex) \wedge$
      $B.LowerBound =$
        $A.LowerBound \wedge B.UpperBound = A.UpperBound \wedge$
      $B.Id = A.Id \wedge \neg B.RestInUse$

  operation ReplaceRest(
      updates A: DistinguishedIndexArray,
      clears B: DistinguishedIndexArray)
    requires
      $A.RestInUse \wedge B.Domain = A.Domain \setminus \{A.DistinguishedIndex\}$
    ensures $A = \#A$ except $\big(\neg A.RestInUse \wedge$
      $EqExceptOn(A.Contents, \#B.Contents, A.DistinguishedIndex)\big)$

  operation ChangeDistinguishedIndexTo(
      updates A: DistinguishedIndexArray,
      restores i: Integer)
    requires $\neg A.RestInUse \wedge \neg A.DistinguishedEntryInUse$
    ensures $A = \#A$ except $A.DistinguishedIndex = i$

```

```
  operation LowerBound(preserves A: SplittableArray):
   Integer
    ensures $LowerBound = A.LowerBound$

  operation UpperBound(preserves A: SplittableArray):
   Integer
    ensures $UpperBound = A.UpperBound$

  operation IsInRest(
      preserves A: DistinguishedIndexArray,
      restores i: Integer): Boolean
    ensures $IsInRest = (i \in A.Domain \land i \neq A.DistinguishedIndex)$

  operation DistinguishedIndex(
      preserves A: DistinguishedIndexArray): Integer
    ensures $DistinguishedIndex = A.DistinguishedIndex$

  operation RestIsInUse(
      preserves A: DistinguishedIndexArray): Boolean
    ensures $RestIsInUse = A.RestInUse$

  operation IdsMatch(
      preserves A1: DistinguishedIndexArray,
      preserves A2: DistinguishedIndexArray): Boolean
    ensures $IdsMatch = (A1.Id = A2.Id)$

end DistinguishedIndexArrayTemplate
```

The model of a `DistinguishedIndexArray` is not dissimilar to that of a `SplittableArray`. The primary difference is that a client only retrieves the "rest" of a `DistinguishedIndexArray`: the entry at the distinguished index is always accessible. The postcondition of `RetrieveRest` guarantees that at any time, there is only one array of a particular `Id` with any given distinguished index.

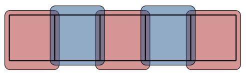
arr = 

Figure 4.7: An `IndexPartitionableArray` can be partitioned into two subarrays of arbitrary index sets.

4.4.2 Index Partitionable Array

The third array abstraction in this family of arrays, called `IndexPartitionableArray`, is a generalization of both the `SplittableArray` abstraction and the `DistinguishedIndexArray` abstraction in section 4.4.1. It is a generalization in the sense that each of the above concepts can be implemented on top of this concept (that is, using an instance of `IndexPartitionableArrayTemplate` it is possible to realize either `SplittableArray` or `DistinguishedIndexArray`). The `IndexPartitionableArray` abstraction itself is useful, for example, in the implementation of heterogeneous parallel operations in which the assignment of data to processes is dynamic. The specification of `IndexPartitionableArray` is in listing 4.6.

Listing 4.6: Specification for `IndexPartitionableArray`.

```
concept IndexPartitionableAraryTemplate(type Entry)

  var Ids: ℘(ℤ)
    initialization ensures Ids = ∅

  type family IndexPartitionableArray is modeled by (
    Id: ℤ,
    LowerBound: ℤ,
    UpperBound: ℤ,
    Contents: ℤ → Entry,
    Red: ℘(ℤ),
    Blue: $\wp(\mathbb{Z})$,
    RedPartInUse: $\mathbb{B}$,
    BluePartInUse: $\mathbb{B}$
  )
  exemplar A
    constraint $A.LowerBound \leq A.UpperBound \wedge$
      $A.Id \in Ids \wedge$
      $A.Red \cap A.Blue = \emptyset$
    initialization ensures
      $A.LowerBound = 0 \wedge A.UpperBound = 0 \wedge$
      $\neg A.RedPartInUse \wedge$
      $\neg A.BluePartInUse \wedge$
      $A.Id \notin \#Ids \wedge$
      $A.Blue = \{(i : \mathbb{Z})(A.LowerBound \leq i \wedge i < A.UpperBound)(i)\}$
  end

  operation SetBounds(
      restores LB: Integer,
      restores UB: Integer,
      updates A: IndexPartitionableArray)
    requires $LB \leq UB \wedge \neg A.RedPartInUse \wedge$
      $\neg A.BluePartInUse$
    ensures $A.LowerBound = LB \wedge A.UpperBound = UB \wedge$
      $\neg A.RedPartInUse \wedge \neg A.BluePartInUse \wedge$
      $\neg A.Id \in \#Ids \wedge \neg \#A.Id \in Ids \wedge$
      $A.Blue \cup A.Red =$
        $\{(i : \mathbb{Z})(A.LowerBound \leq i \wedge i < A.UpperBound)(i)\}$

  operation MakeEntryRed(
      restores p: Integer,
      updates A: IndexPartitionableArray)
    requires $\neg A.RedPartInUse \wedge \neg A.BluePartInUse \wedge$
      $p \in A.Red \cup A.Blue$
    ensures $A = \#A$ except $p \in A.Red$

  operation MakeEntryBlue(
      restores p: Integer,
      updates A: IndexPartitionableArray)
    requires $\neg A.RedPartInUse \wedge \neg A.BluePartInUse \wedge$
      $p \in A.Red \cup A.Blue$
    ensures $A = \#A$ except $p \in A.Blue$

  operation SwapEntryAt(
        restores i: Integer,
        updates A: IndexPartitionableArray,
        updates E: Entry)
      requires $\neg A.RedPartInUse \wedge \neg A.BluePartInUse \;\wedge$
        $A.LowerBound \leq i \wedge i < A.UpperBound \;\wedge$
        $i \in A.Red \cup A.Blue$
      ensures $E = \#A.Contents(i) \;\wedge$
        $A = \#A$ except $A.Contents(i) = \#E$

    operation RetrieveRedPart(
        updates A: IndexPartitionableArray,
        replaces B: IndexPartitionableArray)
      requires $\neg A.RedPartInUse$
      ensures $(A = \#A$ except $A.RedPartInUse) \wedge B.Id = A.Id \;\wedge$
        $B.Red = \#A.Red \wedge B.Blue = \emptyset \wedge B.Contents = \#A.Contents \;\wedge$
        $\neg B.RedPartInUse \wedge \neg B.BluePartInUse \;\wedge$

    operation ReplaceRedPart(
        updates A: IndexPartitionableArray,
        clears B: IndexPartitionableArray)
      requires $A.RedPartInUse \;\wedge$
        $\neg B.RedPartInUse \wedge \neg B.BluePartInUse \;\wedge$
        $B.Id = A.Id \wedge B.Red \cup B.Blue = A.Red$
      ensures $A = \#A$ except
     $\big(A.Contents = \#B.Contents \wedge \neg A.LowerPartInUse\big)$

    operation RetrieveBluePart(
        updates A: IndexPartitionableArray,
        replaces B: IndexPartitionableArray)
      requires $\neg A.BluePartInUse$
      ensures $(A = \#A$ except $A.BluePartInUse) \wedge B.Id = A.Id \;\wedge$
        $B.Blue = \#A.Blue \wedge B.Red = \emptyset \wedge B.Contents = \#A.Contents \;\wedge$
        $\neg B.RedPartInUse \wedge \neg B.BluePartInUse \;\wedge$

    operation ReplaceBluePart(
        updates A: IndexPartitionableArray,
        clears B: IndexPartitionableArray)
      requires $A.BluePartInUse \;\wedge$
        $\neg B.RedPartInUse \wedge \neg B.BluePartInUse \;\wedge$
        $B.Id = A.Id \wedge B.Red \cup B.Blue = A.Blue$
      ensures $A = \#A$ except
     $A.Contents = \#B.Contents \wedge \neg A.LowerPartInUse$

```

```
  operation LowerBound(preserves A: SplittableArray):
   Integer
    ensures $LowerBound = A.LowerBound$

  operation UpperBound(preserves A: SplittableArray):
   Integer
    ensures $UpperBound = A.UpperBound$

  operation RedPartIsInUse(
      preserves A: IndexPartitionableArray): Boolean
    ensures $RedPartIsInUse = A.RedPartInUse$

  operation BluePartIsInUse(
      preserves A: IndexPartitionableArray): Boolean
    ensures $BluePartInUse = A.BluePartInUse$

  operation IsBlue(
      restores i: Integer,
      preserves A: IndexPartitionableArray): Boolean
    ensures $IsBlue = i \in A.Blue$

  operation IsRed(
      restores i: Integer,
      preserves A: IndexPartitionableArray): Boolean
    ensures $IsRed = i \in A.Red$

  operation IdsMatch(
      preserves A1: IndexPartitionableArray,
      preserves A2: IndexPartitionableArray): Boolean
    ensures $IdsMatch = (A1.Id = A2.Id)$

end IndexPartitionableArrayTemplate
```

An `IndexPartitionableArray` provides operations to individually assign indices of the array to either the "red" or "blue" sets, and to retrieve either or both of those sets as (non-contiguous) subarrays. The operation and use of an `IndexParitionableArray` is entirely analogous to the use of a

`SplittableArray` except that indices are individually identified (rather than as a range) to facilitate an arbitrary partition of the array.

4.4.3 Layered Implementations

The `IndexPartitionableArray` (and `DistinguishedIndexArray`) can be efficiently implemented in a similar manner to `SplittableArray`; that is, by sharing a single underlying array amongst all instances with the same `Id`. Once an efficient realization[8] for `IndexPartitionableArray` is provided, however, more specialized realizations can be built by layering on top of it, preserving performance while avoiding additional reasoning costs. For example, `SplittableArray` can be realized with an underlying `IndexPartitionableArray` and mapping the two sets of indices of the `IndexPartitionableArray` to the *low* and *high* parts of the `SplittableArray` and maintaining the invariant that each set of indices in the underlying `IndexPartitionableArray` is contiguous (and corresponds to the appropriate indices for abstraction to a `SplittableArray`).

4.5 Conclusions

The family of arrays presented here provides a basis for a generalizable methodology for designing data abstractions that enable safe parallelism without the need for novel specification constructs and amortize the reasoning costs associated with verifying the correctness of a class of fork-join parallel programs. Programs written with any of these data abstractions can be proven correct with standard client verification machinery, such as tools developed for the verification of RESOLVE programs.

[8]There is no implementation of this kind shown here because the machinery for specifying such shared realizations in RESOLVE is not yet developed and is not a contribution of this dissertation.

However, building and verifying the correctness of an efficient implemenation of any of these abstractions *does* require novel specification constructs, perhaps of the kind introduced in the next chapter, along with more sophisticated reasoning machinery of the kind introduced by Sun [132].

Chapter 5: Abstract Non-Interference Specifications

When concurrent threads of execution do not modify shared data, their parallel execution is equivalent to their sequential execution. For many imperative programming languages, however, the modular verification of this independence is frustrated by (i) the possibility of aliasing between variables mentioned in different threads, and (ii) the lack of abstraction in the description of read/write effects of operations on shared data structures. The reasoning framework introduced in chapter 3 can be implemented to overcome these frustrations.

The non-interference contract is such an implementation for the RESOLVE programming language that permits the specification and (modular) verification of parallel operations on objects in the language. Two examples of software components that are useful in parallel programs are discussed to show the utility of the non-interference contract as a tool to aid reasoning about parallel programs. The first example is that of a bank account with which is is possible to perform a withdrawal concurrently with a deposit. The second example is a classic concurrent data structure, a bounded queue. Three different implementations of a queue are described, each with varying degrees of entanglement and therefore different degrees of possible synchronization-free concurrency.

5.1 Motivating Example: Bank Account Manager

As a motivating example, consider a bank account. This account belongs to a bank that has two “modes” of overdraft handling: (1) no overdrafts, *i.e.*, a withdrawal of an amount greater than the current account balance is never permitted, and (2) any withdrawal is valid so long as, by the end of the day, the account balance is non-negative.

The `BankAccount` type (which has a field *balance* modeled as an unbounded natural number) provides two revelant methods to the client:

- `Withdraw(amt: Natural, acct: BankAccount)`
- `Deposit(amt: Natural, acct: BankAccount)`

5.1.1 Operation Specifications

To reason formally about a program involving objects of type `BankAccount`, the behavior of the operations must be specified. One possible pair of specifications for the operations is as follows.

```
operation Withdraw(preserves amt: Natural, updates acct:
    BankAccount)
  requires $acct.balance \geq amt$
  ensures $acct.balance = \#acct.balance - amt$

operation Deposit(preserves amt: Natural, updates acct:
    BankAccount)
  ensures $acct.balance = \#acct.balance + amt$
```

The reason for the precondition to `Withdraw` is obvious: you can’t withdraw funds you don’t have! However, in mode 2, the bank actually *does* allow such a withdrawal—provided the money is paid back by the end of the day. The question, then, is how to model mode 2 of the bank's operation, and how to prove a prgram written with that model is correct.

5.1.2 Client Programs

Consider the following sequence of transactions on a bank account named `acc`.

Listing 5.1: A sequential client program using a bank account named `acc`.

```
assume acc.balance = 100
-- day begins
Deposit(100, acc)
Withdraw(150, acc)
-- day ends
confirm acc.balance = 50
```

It should be clear that a sequential program such as listing 5.1 models mode 1 of the bank's operation, that is, a withdrawal of more money than is in the account is *never* permitted to take place. (Note that had the two operations in the listing above had been written in the reverse order, the program would not be correct.) Mode 2 can be modeled by considering the day itself as a single transaction period, and the various individual transactions as parallel operations within that period. A valid day in this model is one in which there is *some* serialization of the transactions that never brings the account balance below zero. For example, the client program for the same set of transactions in this model might look like the following.

Listing 5.2: A client program written as a parallel program to model mode 2 of the bank's operation.

```
assume acc.balance = 100
-- day begins
cobegin
  Deposit(100, acc)
  Withdraw(150, acc)
end
-- day ends
confirm acc.balance = 50
```

5.1.3 Implementation

Of course, the client program of listing 5.2 program is deterministic only when the parallel calls are non-interfering. There are several ways to guarantee non-interference. The first approach is to use locks to guarantee exclusive control of shared state by a single process at a time. The second (preferred, for this discussion) approach is to implement the bank account not with a single natural number that is added to or subtracted from within each operation call, but rather as a pair (*in*, *out*) of natural numbers—one that represents the total amount deposited and the other the total amount withdrawn. In this way, a call to `Withdraw(x, a)` is always non-interfering with a call to `Deposit(y, a)` (because they operate on differnt pieces of the underlying data structure). Two important parts of the implementation, the abstraction function (*correspondence* in RESOLVE) and representation invariant (*convention*), are as follows.

```
type BankAccount is (in: Natural, out: Natural)
  exemplar a
  correspondence $a.balance = in - out$
  convention $in \geq out$
```

The abstract model (a single natural number *balance*) is entirely decoupled from the implementation (a pair of natural numbers (*in*, *out*)) except that an implementation provides a mapping from its concrete state to its abstract state. The concrete state of an implementation is determined, in turn, on the *abstract* values of its fields.

Unfortunately, sound reasoning about the parallel program in listing 5.2 involves exposing these implementation details, violating the principles of modularity and abstraction. To overcome this problem, a new specification construct that enables sound modular reasoning about that program and others like it is presented in section 5.3 below.

5.2 Motivating Example: Concurrent Bounded Queue

While there is only one sensible implementation of the bank account type above that permits the desired degree of synchronization-free parallism, other data abstractions invite several reasonable implementations, each of which might differ in the degree of interference-free parallelism they enable. Concurrent queues are one such common data abstraction used in parallel programs. However, most of them employ a form of locking to prevent concurrent access to the head (or tail) of the queue. The queue presented here differs from traditional concurrent queues in several ways: (i) it is designed and specified to avoid the need for locks or other synchronization, provided the queue's size is always middling, (ii) it is bounded (for ease of implementation), and (iii) it is designed to avoid reference leaks and thus permit clean semantics. Recognize that the boundedness is not fundamental to the topic at hand: an unbounded queue could be implemented using the same principles.

5.2.1 Abstract Specification

The `BoundedQueueTemplate` concept models a queue as a mathematical string of "items" (of some arbitrary type); its full specification is in listing 5.3. This concept defines five queue operations that have been designed and specified to avoid the reasoning pitfalls associated with aliasing that arises when a queue contains non-trivial objects [73] and to facilitate clean semantics.

Listing 5.3: Specification for `BoundedQueueTemplate`.

```
concept BoundedQueueTemplate (type Item, MAX_LENGTH: Integer
   )
```

```
  type Queue is modeled by string of Item
    exemplar q
    constraint $|q| \leq MAX\_LENGTH$
    intialization ensures $q = \langle\rangle$

  operation Enqueue (clears e: Item, updates q: Queue)
    requires $|q| < MAX\_LENGTH$
    ensures $q = \#q \circ \langle\#e\rangle$

  operation Dequeue (replaces r: Item, updates q: Queue)
    requires $|q| > 0$
    ensures $\#q = \langle r\rangle \circ q$

  operation SwapFirstEntry (updates e: Item, updates q:
   Queue)
    requires $|q| > 0$
    ensures
      $\langle e\rangle = \#q[0,1)$ $\wedge$
      $q = \langle\#e\rangle \circ \#q[1,|\#q|)$

  operation Length (restores q: Queue) : Integer
    ensures $Length = |q|$

  operation RemCapacity (restores q: Queue) : Integer
    ensures $RemCapacity = MAX\_LENGTH - |q|$

end BoundedQueueTemplate
```

The precondition (**requires** clause) for Enqueue states that there must be space in the queue for the new element. Formally, this is expressed as $|q| < MAX\_LENGTH$. The postcondition (**ensures** clause) says that the outgoing value of q is the string concatenation of the incoming value of q (denoted $\#q$) and the string consisting of a single item, the old value of e. Formally, this is expressed as $q = \#q \circ \langle\#e\rangle$. In simple terms, Enqueue puts the item e at the end of the queue q. The parameter mode (**clears**) for e defines its outgoing value: an initial value for its type (*e.g.*, the value 0 if Item is a numeric type or an empty tree if it is a tree).

The requires clause for `Dequeue` says that q must not be empty—formally, $|q| > 0$. The ensures clause says that the concatenation of the resulting element r and outgoing value of q is the original value of q (*i.e.*, it removes the item from the head of the queue and places it into r)—formally, $\#q = \langle r \rangle \circ q$.

The `SwapFirstEntry` operation allows a client to retrieve the queue's head without introducing aliasing. A traditional concurrent queue might instead use a `Peek` operation that returns an *alias* to the element at the head of the queue.

The functions `Length` and `RemCapacity` behave as expected: `Length` returns an integer equal to the number of elements in the queue, and `RemCapacity` returns an integer equal to the number of free slots left in the queue before it becomes full. Neither changes the abstract value of the queue.

5.2.2 Possible BoundedQueueTemplate Implementations

Three alternative implementations of the bounded queue specified above have been developed, each with different parallelization opportunities. All three are based on an array. In the first two implementations, the length of the underlying array is equal to the maximum length of the queue, MAX_LENGTH, while in the third the length of the array is one greater than MAX_LENGTH.

The first implementation has two integer fields, `front` and `length`, where `front` is the index in the array of the first element of the queue and `length` is the number of elements in the queue. This implementation cannot handle concurrent calls to `Enqueue` and `Dequeue` without synchronization because both of those calls must necessarily change the value of `length`. A client can, however, make concurrent calls to `SwapFirstEntry` and `Enqueue` when the preconditions for both methods

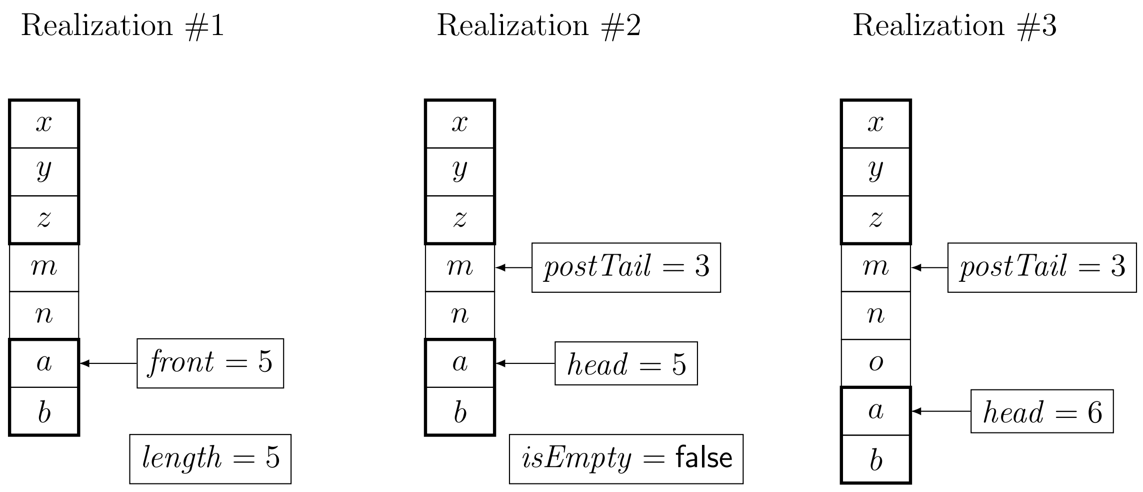


Figure 5.1: Three alternative `BoundedQueueTemplate` implementations on an array, each with the same abstract value $\langle a, b, x, y, z\rangle$.

are met before the parallel block (that is, if $0 < |q|$ and $|q| < MAX\_LENGTH$).[9] These two methods may be executed in parallel because `SwapFirstEntry` touches only the entry at the head of the queue (by reading `front`) and does not even read the value of `length`, while `Enqueue` writes `length` and touches the only the entry just past the tail of the queue (also by reading `front`)—which we know is different from the head of the queue because the precondition of `SwapFirstEntry` guaranteed there was at least one entry in the queue. An empty queue in implemented in this way has $length = 0$ and $0 \leq front < MAX\_LENGTH$, and a full queue has $length = MAX\_LENGTH$ and $0 \leq front < MAX\_LENGTH$.

The second realization also has two integer fields (named `head` and `postTail`) and an additional boolean field `isEmpty`. The field `head` is the index of the array

[9]A more detailed discussion of the proof rule for the parallel composition of statements is found in chapter 3 and below in section 5.4.1; in fact the precondition for parallel composition is more subtle than this.

at which the first element of the queue is located and `postTail` is the index of the first element of the array *after* the last element of the queue. The boolean field `isEmpty` is used to distinguish between a full queue and an empty queue. As in realization #1, a client can concurrently call `Enqueue` and `SwapFirstEntry` as long as both preconditions are satisfied. Additionally, it appears as though it should be possible to concurrently call `Enqueue` and `Dequeue` because the length of the queue is computed on demand from the `head` and `postTail` fields (and not some other field written by both `Engueue` and `Dequeue`). Unfortunately, the obvious implementations of `Enqueue` and `Dequeue` for this queue realization might both attempt to change the value of `isEmpty`. To facilitate the expected degree of parallelism, we can augment the functional specification of `BoundedQueueTemplate` by adding a method: `DequeueFromLong`, which has the same postcondition as `Dequeue`, but has the precondition that the queue will not be either empty nor full after the operation. The formal specification is in listing 5.4. By augmenting the functional specification in this way, a body for `DequeueFromLong` need not check if the queue has been made empty (the precondition precludes its invocation in that situation), and can simply increment the appropriate queue endpoint field.

Listing 5.4: Augmentation of `BoundedQueueTemplate`.

```
operation DequeueFromLong (replaces r: Item, updates q:
   Queue)
  requires |q| − 1 > 0
  ensures #q = ⟨r⟩ ∘ q
```

The third implementation is similar to the second in that its two integer fields are `head` and `postTail` (and they act identically to realization #2), but in lieu of a boolean `isEmpty` field, there is a sentinel element added to the array so that when *head* = *postTail* it can *only* be the case that the queue is empty (a full queue has

$head = (postTail - 1) \mod (MAX\_LENGTH + 1))$. Because the length of the array is greater than MAX_LENGTH, there will always be some element of the array that is not part of the queue. Being able to differentiate between a full and empty queue without the need to have a separate variable ensures that even when the queue might become empty during a call to `Dequeue`, that operation body need not observe or change the value of any variable that `Enqueue` either observes or changes. Therefore, `Enqueue` may be executed concurrently with `Dequeue` when this realization is used (and applicable preconditions are met).

The variety of parallelization opportunities arising from these different implementations of a single interface illuminates the need for a new specification construct that can expose just enough implementation details to enable reasoning about the concurrent execution of several statements for several classes of implementations for a single behavioral specification.

5.3 Non-Interference Contracts and Modular Verification

Modular reasoning about the safe execution of concurrent threads can be separated into three distinct tasks:

(i) a one-time description of the conditions under which operations are independent,

(ii) a proof that client code ensures these independence conditions, and

(iii) a one-time proof that an implementation guarantees non-interference under these conditions.

A functional specification such as in listing 5.3 by necessity does not reveal the degree to which different parts of the concrete state are entangled in the implementation. The correspondence relation between concrete state and abstract state *does* reveal the degree of entangledness and is part of the proof of correctness for the implementation, but the principles of modular verification preclude its use in reasoning about client code.

Reasoning soundly about the independence of concurrent threads in client code, however, requires exposing some implementation details. One approach for describing this independence involves defining an intermediate model consisting of orthogonal components, and encapsulating the description of this intermediate model in a distinct specification construct, a *non-interference contract.* While a simple, static segmentation is all that is necessary for the examples here, in general a non-interference contract could introduce a dynamic number of orthogonal components or annotations for synchronization operations.

The relationships of a non-interference contract to other parts of a software component are illustrated in fig. 5.2. Its use is explored by applying it to the two examples from above (sections 5.1 and 5.2).

5.3.1 Describing Non-Interference

The non-interference contract specification construct partitions the representation space of a data type into a number of *pieces*, each of which is disjoint from the others in the sense that at the representation level, each piece of data (*e.g.*, each representation field[10]) is a member of exactly one piece. A non-interference contract

[10]Notice that because non-interference contracts are modular, a representation field might have a piece of its partition—not necessarily its whole self—that maps to the higher-level partition.

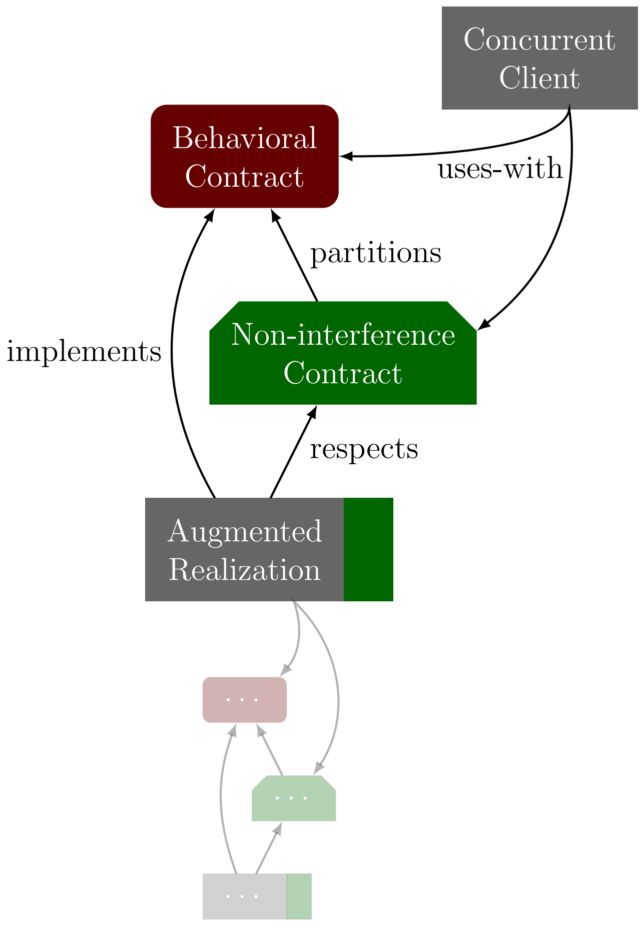


Figure 5.2: Component diagram illustrating the role of non-interference contracts.

extends an abstract specification to include effects summaries that define how an operation interacts with the pieces of an object's partition. These summaries are described in terms of three modes: *affects*, *preserves*, and *ignores*. In addition to these three modes, an operation is said to have a *restructures* effect if, by executing the operation, the members of some piece of a parameter's partition might be moved to a different one, though none of the *values* of those members will have changed. All three modes and the restructures effect may be conditional on the abstract values of the parameters.

A *when clause* gives a condition that restricts the scope of effects by defining a predicate that must hold initially for the stated effect to occur. When clauses help to generalize non-interference contracts so that potentially many implementations could satisfy a single non-interference contract that exposes desirable parallelism properties. Reasoning about when clauses relies on an abstract behavioral specification and the ability to reason about the abstract specification; this is a power of using a programming language with sequential verification capabilities such as RESOLVE as a basis for verification of parallel programs.

Non-interference contracts are designed to ease reasoning about cobegin blocks and the rules for showing the non-interference of several statements inside a cobegin block in RESOLVE code using interference contracts are relatively straightforward. Each piece of the partition of an object mentioned in a cobegin block's constituent statements may be *affects*-mode in at most one statement, and if it is *affects*-mode in some statement then it must be *ignores*-mode in all others. When these properties are satisfied, the statements in the cobegin block are said to be *non-interfering* and their execution will be deterministic. Once non-interference (and therefore determinism) is established using the non-interference contract, verification can proceed as if the execution of the operations were sequential. In terms of the A/P Calculus and framework from chapter 3, the specified effect of each statement in the cobegin block must be non-interfering with the sepcified effect of every other statement in the block.

RESOLVE's clean semantics ensure that an operation *ignores* (*i.e.*, neither reads nor writes) any variable not explicitly included as a parameter. Similarly, an operation ignores any piece of a partition not explicitly mentioned in its effects.

The Restructures Effect

In component implementations that make use of dynamically assigned partitions, a subtle interaction can occur: an operation might shuffle the membership of an object's partition without altering the values of any fields that live in its pieces. In this case, we say that method *restructures* some pieces of that partition. The restructures effect is fundamentally different than three partition modes in that it places no proof obligation on the implementation, but it does place an obligation on the client. If a statement in a program restructures pieces of an object's partition, all *subsequent* statements in that program must have those pieces in *ignores*-mode.

5.4 Reasoning About Non-Interference Between Threads

In the programs examined in this dissertation, non-interference comes in several flavors. First, processes are non-interfering if they share no variables in common (such as in chapter 4). Second, processes are non-interfering if they do share a variable but it is only read from—not written to—by both processes. Finally, processes can be non-interfering even when they share and modify a variable through the use of non-interference contracts.

In addition to satisfying the usual preconditions for functional correctness, the verification of the client code includes establishing the conditions of non-interference of the two operations. The establishment of these conditions is carried out entirely in the context of the client code, using only the abstract behavioral specification and non-interference contract of the type.

5.4.1 A Modular Proof Rule for Parallel Operation Calls

Given a set of operations with some behavioral specifications that have been augmented with effects summaries (*e.g.*, in a non-interference contract), the semantics of their parallel composition is as in eq. (5.1). The specific rule is given only for the two-way parallel composition of a pair of operations A and B (with specifications given in listing 5.5), though it could readily be generalized to n arbitrary parallel operation calls.

Listing 5.5: Specifications of two general operations for formulating a proof rule for parallel operation calls.

```
partition for T2 is {a, b}

operation A(preserves x: T1, updates y: T2)
    preserves x
    affects y@a
    when G_A(x,y) affects y@b
    requires Pre_A(x,y)
    ensures Post_A(x,#y,y)

operation B(preserves x: T1, updates y: T2)
    preserves x
    affects y@b
    when G_B(x,y) affects y@a
    requires Pre_B(x,y)
    ensures Post_B(x,#y,y)
```

$$\frac{\begin{array}{l}(\mathsf{pre}_{A;B}) \xrightarrow{A(u,v);B(u,v)} (\mathsf{post}_{A;B}) \\ \wedge\, (\mathsf{pre}_{B;A}) \xrightarrow{B(u,v);A(u,v)} (\mathsf{post}_{B;A})\end{array} \qquad G \Rightarrow A(u,v) \ddagger B(u,v)}{\big(G \wedge (\mathsf{pre}_{A;B} \vee \mathsf{pre}_{B;A})\big) \xrightarrow{A(u,v) \,\|\, B(u,v)} (\mathsf{post}_{A;B} \wedge \mathsf{post}_{B;A})} \tag{5.1}$$

Here, G is a predicate on the abstract values of the variables u and v that must be true for the two statements to be non-interfering (notated with ‡). In this case, $G = \neg G_A(u,v) \wedge \neg G_B(u,v)$. The predicate $\mathsf{pre}_{A;B}$ (respectively, $\mathsf{pre}_{B;A}$) is the composite precondition for the serialization $A(u,v); B(u,v)$ (respectively, $B(u,v); A(u,v)$)—that is, the weakest predicate that, when satisfied, will produce a trace of those operation calls with no violated preconditions. $\mathsf{post}_{A;B}$ and $\mathsf{post}_{B;A}$ are the composite postconditions of the serializations.

5.4.2 Verifying the Respects Relation for an Implementation

A mapping from concrete implementation state to abstract specification state is provided by a realization's representation invariant (called a *convention* in RESOLVE) and abstraction relation (*correspondence*). To establish operation non-interference, the correspondence relation is augmented with a partitioning of the constituent concrete state space. Thus, a realization must provide a mapping from the concrete data structure involved in the implementation to the partitioned model of the non-interference contract. Specifically, it must place each piece of the realization into exactly one of the pieces of the non-interference contract.

The proof obligation for a realization is as follows: the actual effect of the body of each operation must be covered by the specified effect in the non-interference contract, under the conditions specified in any when clauses. An operation body ignores a piece of a partition only when all statements in its body ignore the corresponding parts of the data structure. The underlying data structure itself might be built from other data abstractions with their own non-interference contracts. This is not a problem—in fact it enables the non-interference contract to preserve modularity—because the lack of entanglement of one component can be layered on top of appropriately disentangled realization fields, and the actual effect of the body can be computed using the specified effects of its constituent statements.

5.5 Continued Example: Bank Account Manager

A non-interference contract for the operations `Withdraw` and `Deposit` that permits their parallel execution is below.

Listing 5.6: Non-interference contract for type `BankAccount`.

```
noninterference contract TotalIndependence for BankAccount

  partition for BankAccount is {w, d}

  operation Withdraw(preserves amt: Natural, updates acct:
   BankAccount)
    affects acct@w

  operation Deposit(preserves amt: Natural, updates acct:
   BankAccount)
    affects acct@d

end TotalIndependence
```

5.5.1 Reasoning about a Bank Account Client Program

Consider the example parallel client program in listing 5.2 (reproduced here).

```
assume a.balance = 100
cobegin
```

```
  Deposit(100, a)
  Withdraw(150, a)
end
confirm a.balance = 50
```

By the proof rule in eq. (5.1), either (any) of the preconditions for the serializations must be met in order to conclude both (all) of the postconditions. This can result in surprising behavior. Applying the various rules for sequential composition of statements, we have the following two formulas (eqs. (5.2) and (5.3)) that describe the semantics of each serialization.

$$(a.balance \geq 150) \xrightarrow{\texttt{Withdraw(150,a);Deposit(100,a)}} (a.balance = \#a.balance - 50) \tag{5.2}$$

$$(a.balance \geq 50) \xrightarrow{\texttt{Deposit(100,a);Withdraw(150,a)}} (a.balance = \#a.balance - 50) \tag{5.3}$$

When `Deposit` and `Withdraw` are non-interfering, as is the case when we use an implementation that respects the non-interference contract for `BankAccount` from listing 5.6, the semantics of their parallel execution as defined by eq. (5.1) is as follows.

$$(a.balance \geq 50) \xrightarrow{\texttt{Withdraw(150,a)} \parallel \texttt{Deposit(100,a)}} (a.balance = \#a.balance - 50) \tag{5.4}$$

Notice the precondition for the parallel composition (5.4) is *weaker* than the precondition for one of the serializations (5.2). This is surprising because at runtime the execution of `Withdraw` might be scheduled before the execution of `Deposit`, resulting in a violated precondition if $50 \leq a.balance < 150$. However, because of the non-interference of these two statements, that does not matter: there will still be no data races and the behavior will be identical to that if `Deposit` had come first. Therefore, we can reason about their parallel composition as if the scheduler makes an "angelic" choice: the serialization `Deposit(100, a); Withdraw(150, a)`, in which no preconditions are violated.

This conclusion is achieved with the help of a substantial amount of meta-information that is gleaned from the non-interference of several operations—that is, that the two operations always commute. Without the notion that the statements `Withdraw(150, a)` and `Deposit(100, a)` are non-interfering and thus always commute, their parallel composition with the weak precondition could not be verified as correct because of the precondition for `Withdraw`. Their non-interference means that this specification is artificially weak: there is no operational reason in the body of `Withdraw` for the precondition. That is not to say that an artificially weak specification is useless—many problems, in fact, rely on weak specifications for generality. In the example at hand, however, the relatively weak specification of `Withdraw` is a consequence of the abstract view of a `BankAccount`'s balance being a natural number (*i.e.*, a non-negative integer) even though the particular class of implementations under discussion (*i.e.*, those that respect the non-interference contract `TotalIndependence` in listing 5.6) does not rely on that being the case.

In the implementation discussed here, the correspondence and the convention together ensure that the balance is always a natural number. The usual rules for the convention (*i.e.* representation invariant) state that it may only be broken *within* the body of an operation, and must be returned to a valid state before that operation's body ends. However, when parallelism is involved, an operation might be interrupted by another one mid-execution. In this situation, that interrupting operation might begin in a state with a violated convention! However, one interpretation of the definition of non-interference is that the *way* in which a convention might be violated by one operation cannot affect the execution of any non-interfering operation.[11]

5.6 Continued Example: Concurrent Bounded Queue

Listing 5.7 below is a possible non-interference contract for the bounded queue abstraction specified in listing 5.3. It matches the parallelization capabilities provided by realization #1 in fig. 5.1.

Listing 5.7: A non-interference contract for type BoundedQueueTemplate.

```
noninterference contract LookupOffset
  for BoundedQueueTemplate

  partition for Queue is {head, tail, offset}

  operation Enqueue (clears e: Item, updates q: Queue)
    affects q@tail, e@⋆
    preserves q@offset
    when |q| = 0 affects q@head

  operation Dequeue (replaces e: Item, updates q: Queue)
    affects q@head, q@offset, e@⋆
    restructures q@head, q@tail

  operation SwapFirstEntry (updates e: Item, updates q:
   Queue)
    affects q@head, e@⋆
    preserves q@offset

  operation Length (restores q: Queue) : Integer
    preserves q@head, q@offset

  operation RemCapacity (restores q: Queue) : Integer
    preserves q@head, q@offset

end LookupOffset
```

[11]The validity of this interpretation is a consequence of the results from chapter 3.

Names for different pieces of the intermediate model are introduced on line 4 with the *partition* keyword. For example, in the non-interference contract `LookupOffset` partitions the concrete space of a queue into three pieces: $q@head$, $q@tail$, and $q@offset$. The operations `SwapFirstEntry` and `Enqueue` affect distinct pieces ($q@head$ and $q@tail$, respectively). Furthermore, each ignores the piece affected by the other, assuming the queue is non empty. Finally, the piece used by both ($q@offset$) is preserved by both. The following client code illustrates a parallel composition of these operations.

```
assume 0 < |q| < MAX_LENGTH
cobegin
  SwapFirstEntry(x, q)
  Enqueue(y, q)
end
```

First, note that the client code above can be safely executed only if there is no aliasing between objects x and y. This can be achieved by using a programming lanaugage like RESOLVE (that has clean semantics) or through disciplined programming of the kind discussed in chapter 6 to avoid unintended aliasing.

Listing 5.8 below is an (augmented) implementation of `BoundedQueueTemplate` that respects the non-interference contract `LookupOffset` of listing 5.7. Line 24 introduces the *non-interference correspondence*, which identifies how the implementation state is partitioned.

Listing 5.8: Example code showing a realization of `BoundedQueueTemplate` in line with the picture of realization #1.

```
1 realization ArrayWithLength for BoundedQueueTemplate
2   respects LookupOffset
3   uses ArrayTemplate with SeparateElements
4 
5   function FunctionToStringWithWrapping(f: integer -> Entry,
      s: integer, e: integer): string of E is
```

```
    if $s = e$ then $\langle\rangle$
    else $\langle f(s)\rangle \circ \mathit{FunctionToStringWithWrapping}($
                              $f, (s+1) \mod (\mathit{MAX\_LENGTH}), e)$

type representation for Queue is
    (contents: array 0..MAX_LENGTH - 1 of Item,
     front: Integer,
     length: Integer)
exemplar q
    convention
        $0 \leq q.\mathit{front} < \mathit{MAX\_LENGTH} \wedge$
        $0 \leq q.\mathit{length} \leq \mathit{MAX\_LENGTH}$
    correspondence function is
        $\mathit{FunctionToStringWithWrapping}($
            $q.\mathit{contents},$
            $q.\mathit{front},$
            $q.\mathit{front} + q.\mathit{length} + 1$
        $)$
    noninterference correspondence for LookupOffset
        q@head: q.contents@c[q.front]
        q@tail: q.length, q.contents@c except on {q.front}
        q@offset: q.front
end Queue

procedure Enqueue(clears e: Item; updates q: Queue)
    var newIdx := q.front + q.length mod MAX_LENGTH
    e :=: q.contents[newIdx]
    q.length := q.length + 1
    Clear(e)
end Enqueue

procedure Dequeue(replaces e: Item, updates q: Queue)
    e :=: q.contents[q.front]
    q.front := q.front + 1 mod MAX_LENGTH
    q.length := q.length - 1
end Dequeue

procedure SwapFirstEntry(updates e: Item; updates q: Queue
  )
    e :=: q.contents[q.front]
end SwapFirstEntry

function Length(preserves q: Queue)
```

```
    Length := q.length
  end Length

  function RemCapacity(preserved q: Queue)
    RemCapacity := MAX_LENGTH - q.length
  end RemCapacity

end ArrayWithLength
```

Applying the Rules for the Restructures Effect

Observe that the operation `Dequeue` from listing 5.7 *restructures* the pieces *q@head* and *q@tail*. To understand what this entails, consider realization #1 of `BoundedQueueTemplate` as illustrated in fig. 5.1 (the code for which appears in listing 5.8) and in particular how the `Dequeue` operation is implemented. The interference correspondence of that realization places the element of the array at index `q.front` in the *q@head* piece of the object's partition, and the rest of the array in the *q@tail* piece. The body of `Dequeue` in that realization swaps the element of the array at index `q.front` with the parameter `e` (line 38), then increments `q.front` (line 39). Upon changing the value of `q.front`, the operation body has changed the membership of the *q@head* and *q@tail* pieces of `q`'s partition. Therefore, a client of `Dequeue` no longer knows after the operation returns how `q` is partitioned (because the client has no knowledge of the `front` field) and there might be a parallel operation that was non-interfering before the call, but that is no longer non-interfering afterward.

5.6.1 Layering Non-Interference Contracts

Real-world software components are built with many layers of abstraction. The use of a non-interference contract to reason about safe parallelism enables such modular composition, as seen in listing 5.8. Line 3 in the listing identifies that the `ArrayTemplate` instantiation used in this realization uses the `SeparateElements` non-interference contract. That non-interference contract is not formally specified here, but it exposes the expected concurrency properties of an array, that is, it permits the concurrent access to elements of the array at distinct indices (which, by clean semantics, are guaranteed not to be aliases to a shared object). In general, layering non-interference contracts limits the realizations that can be used to instantiate the concept.

Notice that the partition assignment of the field `q.contents` on lines 25 and 26 involves the non-interference contract for the array (*i.e.*, $q.contents@c$ refers to the piece of the partition of `q.contents` named c). It is the partition at this nested level that is used in the realization's interference correspondence.

Proving the Respects Relation

The proof that Enqueue ignores $q@head$ (when the queue is non-empty—otherwise it affects $q@head$ and this proof is trivial) is seen as follows. When the queue is non-empty, $q.length \geq 1$. So the part of `q.contents` that is modified is not `q.contents[q.front]`. Because $q@head$ contains only $q.contents@c[q.front]$, Enqueue ignores $q@head$.

The`SwapFirstEntry` operation, on the other hand, ignores $q@tail$. This can be seen immediately from two facts: that the operation body does not mention `q.length` and that only the entry of `q.contents` at index `q.front` is affected, so it ignores the rest of `q.contents`, and therefore the entirety of $q@tail$.

The proof that $q@offset$ is preserves-mode in both `Enqueue` and `SwapFirstEntry` amounts to a proof that no statement in those bodies writes to the field `q.front` (as that is the only part of the realization that is in the piece named *q@offset*). This proof follows immediately from the effects of the statements in the bodies of `Enqueue` and `SwapFirstEntry`.[12] In particular, the statements that mention `q.front` (line 31 and line 44) only read its value and do not change it.

5.6.2 Other Realizations

Other realizations of `BoundedQueueTemplate` demand different augmentations and implementation annotations for concurrent execution. For example, listing 5.9 is a more permissive non-interference contract that permits concurrent execution of `Enqueue` with either `Dequeue` or `SwapFirstEntry` when the queue is non-empty and non-full. A queue with this non-interference contract might be used in a producer-consumer context when the queue is known never to be empty.

Listing 5.9: A permissive non-interference contract for `BoundedQueueTemplate`.

```
noninterference contract ParEnqDeq
  for BoundedQueueTemplate

  partition for Queue is {a, b}

  operation Enqueue(updates q: Queue, clears e: Entry)
    affects q@b
    affects e@⋆
    when |q| = 0
      affects q@a

  operation Dequeue(updates q: Queue, replaces e: Entry)
    affects q@a
    affects e@⋆
    restructures q@a, q@b

  operation SwapFirstEntry(updates q: Queue, replaces e:
   Entry)
```

[12]Although those effects are not formally specified here, they are as one would expect.

```
    affects q@a
    affects e@*

  function Length(preserves q: Queue): Integer
    preserves q@a, q@b

  function RemCapacity(preserves q: Queue): Integer
    preserves q@a, q@b

end ParEnqDeq
```

The non-interference contract `ParEnqDeq` in listing 5.9 stands out because there are several reasonable implementations that respect it. For example, the `DistinguishedIndexArray` abstraction from section 4.4.1 can be used to implement `BoundedQueueTemplate` in a way that respects this non-interference contract. Code for that realization is in listing 5.10. In the realization the non-interference contract for `DistinguishedIndexArray` has three pieces: *dist*, *rest*, and *idx* (they are used in the noninterference correspondence on lines 31 to 35).

Listing 5.10: A realization of `BoundedQueueTemplate` using an instance of `DistinguishedIndexArray`.

```
1  realization DistinguishedIndexArrayRealization
2    implements BoundedQueueTemplate
3    respects ParEnqDeq
4    uses DistinguishedIndexArrayTemplate with DistIdxSeparate
5
6    function FunctionToStringWithWrapping(
7        f: integer -> Entry,
8        s: integer,
9        e: integer): string of Entry is
10     if $s = e$ then $\langle\rangle$
11     else
12       $\langle f(s)\rangle \circ \mathit{FunctionToStringWithWrapping}($
13                            $f, (s+1) \mod (\mathit{MAX\_LENGTH}+1), e)$
14
15   lemma $\forall(f : \mathbb{Z} \to \mathit{Entry}, s : \mathbb{Z}, e : \mathbb{Z})$
16     $((|\mathit{FunctionToStringWithWrapping}(f, s, e)| = 0) \Leftrightarrow (s = e))$
17
```

```
representation for Queue is
 (contents: DistinguishedIndexArray,
  postTail: Integer)
exemplar Q
  convention
    $0 \leq q.postTail \leq MAX\_LENGTH \;\wedge$
    $q.contents.domain = \{(i:\mathbb{Z})(0 \leq i \wedge i \leq MAX\_LENGTH)(i)\}$
  correspondence function is
    $FunctionToStringWithWrapping($
      $q.contents.contents,$
      $q.contents.distinguishedIndex,$
      $q.postTail$
    $)$
  noninterference correspondence
    q.contents@dist in q@a
    q.contents@rest in q@b
    q.contents@idx in q@a
    q.postTail in q@b
  initialization
    SetBounds(q.contents, 0, MAX_LENGTH)
end Queue

operation Enqueue(updates q: Queue, clears e: Entry)
  if (IsInRest(q.contents, q.postTail))
    var A: DistinguishedIndexArray
    RetrieveRest(q.contents, A)
    ChangeDistinguishedIndexTo(q.postTail, A)
    SwapDistinguishedEntry(A, e)
    ReplaceRest(q.contents, A)
  else
    SwapDistinguishedEntry(q.contents, e)
  end if
  q.postTail := (q.postTail + 1) mod (MAX_LENGTH + 1)
  Clear(e)
end Enqueue

operation Dequeue(updates q: Queue, replaces e: Entry)
  SwapDistinguishedEntry(q.contents, e)
  var Spec: Integer
  Spec := DistinguishedIndex(q.contents)
  Spec := (Spec + 1) mod (MAX_LENGTH + 1)
  ChangeDistinguishedIndexTo(q.contents, Spec)
end Dequeue
```

```

  operation SwapFirstEntry(updates q: Queue, replaces e:
   Entry)
    SwapDistinguishedEntry(q.contents, e)
  end SwapFirstEntry

  operation Length(preserves q: Queue): integer
    Length := q.postTail - DistinguishedIndex(q.contents)
    Length := Length mod (MAX_LENGTH + 1)
  end Length

  operation RemCapacity(preserves q: Queue): integer
    var Length: Integer
    Length := Length(q)
    RemCapacity := MAX_LENGTH - Length
  end RemCapacity

end Distinguished_Index_Array_Realization
```

The non-interference contract `ParEnqDeq` could also be realized by using two instances of `BoundedQueueTemplate`, each of which also respect the `ParEnqDeq` non-interference contract, the code for which is in listing 5.11. (In listing 5.11, the function *intrl* on line 23 in the correspondence is the element-wise interleaving of the two string arguments, beginning with the first argument.)

Listing 5.11: Another realization of `BoundedQueueTemplate` layered on top of two other instances of `BoundedQueueTemplate` that respects the non-interference contract `ParEnqDeq`.

```
 1 realization SplitQueues
 2   implements BoundedQueue
 3   respects ParEnqDeq
 4
 5   uses BoundedQueueTemplate with ParEnqDeq
 6   uses Mod for UnboundedIntegerFacility
 7   uses Increment for UnboundedIntegerFacility
 8   uses Add for UnboundedIntegerFacility
 9
10   facility QueueFacility is BoundedQueueTemplate(Item,
      MAX_LENGTH/2)
11
```

```
type representation for Queue is (queueT: Queue,
                                  queueF: Queue,
                                  nextHead: Boolean,
                                  nextTail: Boolean)
  exemplar q
  convention
    $\big||q.queueT| - |q.queueF|\big| \leq 1 \ \wedge$
    $|q.queueT| < |q.queueF| = (\neg q.nextHead \wedge q.nextTail) \ \wedge$
    $|q.queueT| > |q.queueF| = (q.nextHead \wedge \neg q.nextTail) \ \wedge$
    $(|q.queueT| = |q.queueF|) = (q.nextHead = q.nextTail)$
  correspondence function is
    $\begin{cases} intrl(q.queueT, q.queueF) & q.nextHead \\ intrl(q.queueF, q.queueT) & \texttt{otherwise} \end{cases}$
  noninterference correspondence for ParEnqDeq
    q.nextHead is in q@a
    q.nextTail is in q@b
    q.queueT@a is in q@a if q.nextHead, else in q@b
    q.queueF@a is in q@a if not q.nextHead, else in q@b
    q.queueT@b is in q@b
    q.queueF@b is in q@b
end Queue

procedure Enqueue (clears e: Item, updates q: Queue)
  if (q.nextTail) then
    Enqueue(e, q.queueT)
  else
    Enqueue(e, q.queueF)
  end
  Negate(q.nextTail)
end Enqueue

procedure Dequeue (replaces r: Item, updates q: Queue)
  if (q.nextHead) then
    Dequeue(r, q.queueT)
  else
    Dequeue(r, q.queueF)
  end
  Negate(nextHead)
end Dequeue

procedure SwapFirstEntry (updates e: Item, updates q:
 Queue)
```

```
    if (q.nextHead) then
      SwapFirstEntry(e, q.queueT)
    else
      SwapFirstEntry(e, q.queueF)
    end
  end SwapFirstEntry

end SplitQueues
```

5.7 Conclusions

Although the semantics for the parallel programs have been well-studied, essentially all prior work on verifying the correctness of parallel programs involves reasoning about explict implementation details of the software components that are used. The non-interference contract is a novel specification construct that works in concert with a behavioral specification to expose only some implementation details to enable the safe parallel composition of several operations on an object and preserves the ability to soundly reason in a modular fashion about parallel software. By partitioning the representation space of a type into a number of pieces and defining how various operations interact with those pieces, a non-interference contract can guarantee that some parallel programs will be data race free even if parallel statements share abstract variables and do not involve explicit synchronization constructs. As a consequence of these guarantees, a verifer may reason about non-interfering parallel programs as if they were sequential, paving the way for the automated verification of a variety of parallel programs that make use of high-level abstract data types.

Chapter 6: Using C++ Move Semantics to Minimize Aliasing and Simplify Reasoning

Efforts to ease reasoning about software written in RESOLVE and other academic programming languages are valuable endeavors, but are not immediately adoptable by the wider software engineering community because of the languages used. The facilitate the transfer of ideas and techniques from academic languages to mainstream programming languages, this chapter presents a discipline for C++ programming that produces programs that exhibit many of the properties of RESOLVE programs that make the language an attractive target for automated formal verification.

Most modern programming languages rely on pointer and reference copying for efficient data movement. When references to mutable objects are copied, aliases are introduced, often complicating formal and informal behavioral reasoning. While some aliasing (and the resulting increase to reasoning complexity) is generally regarded as necessary in performant software, aliasing can be minimized for practical software development by leveraging the relatively recent introduction of move semantics in C++ (as of C++11), while also satisfying important performance requirements of real-world software. Applying a carefully-developed discipline for programming based on move semantics can avoid most routine introduction of aliasing in programs, thereby leading to simpler reasoning about the behavior of software. The discipline, called

Clean++, requires attention to interface and implementation design, and it is illustrated through a series of components that have been developed to address a variety of programming use cases.

6.1 Move Semantics in C++

Introduced in the C++11 specification, move semantics is a mechanism designed to improve the performance of data movement [78]. It is an efficient alternative to both traditional value and reference copying. Its efficiency stems from avoiding a deep copy by *moving* a value from one variable to another, leaving the old variable in an undefined state.

In C++, an expression appearing on the right-hand side of an assignment operation is one of two kinds: an *lvalue* or an *rvalue*. An lvalue represents something occupying an identifiable location in memory and is therefore suitable for the left-hand side of an assignment, for example a variable (`x`), a pointer dereference (`*p`), or a function returning a reference (`a[3]`). An rvalue, on the other hand, represents the temporary result of an expression evaluation and is not suitable for the left-hand side of an assignment, for example the result of a constructor (**`new`** `T()`), a literal (`"Hello"`), or an address-of operation (`&x`). Beginning with C++11, rvalues are further divided into *rvalue references* and *const rvalues*: the former may be modified while the latter may not.

What use is it to modify an rvalue, given that rvalues are temporary values poised to go out of scope very soon anyway? A key observation that led to the inclusion of move semantics is that in order to implement "moving assignment", the right-hand side might have to be modified (*e.g.*, by being set to `nullptr` or an initial value for its type). The idea behind moving assignment is that precisely *because* the right-hand side is poised to go out of scope, the left-hand side does not need to make a copy. Rather, it is enough to merely steal ownership of the data from the right-hand side.

Move semantics offer clear performance benefits. When a parameter to a method is passed by value, or when a variable is assigned with the assignment operator, the copy constructor or copy assignment operator is called (which might make an expensive deep copy). For large or complex data types, copying can be expensive or difficult to implement, or both. If the value of an actual parameter to a method call is not needed after the call completes, then parameter passing can be made far more efficient by *moving* the value of the actual parameter to the formal parameter, leaving the actual parameter in an easily constructed (or undefined) state. In [13], a sample program is shown in which copy semantics leads to the creation of several unnecessary copies of a resource; employing move semantics eliminates this copying and results in a single allocation of the resource, whose value is then moved between variables, parameters, and return values.

6.1.1 Reasoning Benefits of Move Semantics

In typical C++ programs, there are often many pointers and references. Sometimes, the use of pointers stems from the implementation of a linked data structure with unavoidable aliases, such as a directed graph. Often, however, aliases arise from efficiency concerns as they provide a mechanism for constant-time and constant-space assignment and parameter passing. For example, consider the `list` component from the C++ standard template library [21]. This component's `insert` method creates a copy of each inserted item. To avoid expensive copying, a `list` could be declared to hold pointers to items (`T*`) rather than the items themselves (`T`).

Unfortunately, the use of pointers and reference semantics introduces significant challenges for reasoning about the behavior of code. At the root of these challenges is the fact that aliases permit a program statement to affect variables that are not explicitly mentioned by that statement (making local reasoning unsound). Put another way, understanding the effect of a program statement requires, in the worst case, a whole-program analysis of pointer variables and the memory addresses they identify.

Listing 6.1: Short `main` method illustrating the reasoning challenges introduced by aliased references.

```
 1 int main(int argc, const char* argv[])
 2 {
 3   int* a = new int[3];
 4   for (int i = 0; i < 3; i++) {
 5     a[i] = i+1;
 6   }
 7   int* b = a;
 8   b[1] = 4; // modify array b
 9   printf("b␣=␣{%d,␣%d,␣%d}\n", b[0], b[1], b[2]);
10   printf("a␣=␣{%d,␣%d,␣%d}\n", a[0], a[1], a[2]);
11   return 0;
12 }
```

Consider the aliasing illustrated in listing 6.1 above. The program prints the same value for both `a` and `b`: `{1, 4, 3}`, even though `a` appears not to be modified after it is initialized in the loop to hold the value `{1, 2, 3}`. Although this simple example is easy to reason about, understanding even slightly more complicated programs in a systematic, modular, or automated way is intractable in the general case (for example because some code that modifies `b` might be hidden inside the body of a function or method call replacing line 8).

Memory Management

One specific manifestation of these challenges is the difficulty of memory management. Properly balancing memory allocation and deallocation is notoriously hard, as evidenced by the ubiquity of errors related to memory leaks, dangling pointers, and null dereferences (and by the existence of garbage collectors). Using move semantics and programming according to a strict discipline with can eliminate many of these errors. Rust [89] is one attempt to tackle this class of errors at the language level by using moving semantics as the default data movement operation and by providing a strict set of rules for data that is shared between variables. It is discussed in some detail in section 2.5.3.

To eliminate aliasing in the example above, we could encapsulate an array inside a class and override the constructor and assignment operator for that class, allowing only the moving assignment and initialization operations. The modified code, including the implementing class, might look like the following listing.

Listing 6.2: Implementation and use of a moving-only array type.

```
class MoveArrInt
{
private:
  int m_a[];
public:
  MoveArrInt(const MoveArrInt& m_arr) = delete;
  MoveArrInt(MoveArrInt&& m_arr)
  {
    m_a = m_arr.m_a;
    m_arr.m_a = new int[0];
  }
  MoveArrInt(int*&& arr)
  {
    m_a = arr;
    arr = new int[0];
  }

  MoveArrInt& operator=(const MoveArrInt& m_arr) = delete;
  MoveArrInt& operator=(MoveArrInt&& m_arr)
  {
    if (&m_arr == this)
    {
      return *this;
    }
    delete[] m_a;
    m_a = m_arr.m_a;
    m_arr.m_a = new int[0];
    return *this;
  }

  int& operator[](std::size_t idx)
  {
    return m_a[idx];
  }
};

int main(int argc, const char * argv[])
{
  MoveArrInt a(new int[3]);
  for (int i = 0; i < 3; i++) { a[i] = i+1; }
  MoveArrInt b = std::move(a);
  b[1] = 4; // modify array b
  printf("b␣=␣{%d,␣%d,␣%d}\n", b[0], b[1], b[2]);
  printf("a␣=␣{%d,␣%d,␣%d}\n", a[0], a[1], a[2]);
  return 0;
}
```

The MoveArrInt class deletes the copy constructor and copy assignment operator, opting instead to only allow construction and assignment when the argument or right-hand side is an rvalue reference. The result, as seen in the main method, is that when assigning a MoveArrInt from another one, the client must enclose the right-hand side in a call to std::move. Introduced in C++11, the std::move operation converts its argument to an rvalue reference, effectively marking it as unowned (and hence modifiable). Because the C++ language specification does not define the value of a variable after it is moved, a moved variable should never be used after such a call until it is assigned a new value. Indeed, printing the value of `b` on line 43 displays `{1, 4, 3}` as in listing 6.1, but printing the value of `a` on line 44 displays garbage values.

In the modified program, there is never more than one variable that "owns" any chunk of memory. Aliasing is eliminated, and it becomes possible to reason locally about the program: the value of `a` is not changed by the statement on line 42, nor can *any* statement not explicitly mentioning `a` change its value.

This design pattern can be extended to include a wide range of data types, including linked data types and others whose values are typically allocated and managed on the heap. By implementing move constructors and move assignment operators and also deleting their copying counterparts, aliases can be prevented while the performance benefits afforded by reference semantics can be preserved.

6.2 Clean++: A Discipline for Software Engineering in C++

The use of move semantics makes it possible to write clean, modular code that is easy to understand and reason about. A discipline based on move semantics can encourage abstraction and understandability, and would guarantee the soundness of local reasoning. The development of such a discipline, called Clean++, is directed by several broad goals:

- *Soundness of Local Reasoning.* A Clean++ program should exhibit behavior that is immediately apparent from locally reasoning about the code: *e.g.*, reasoning about a function call should not require knowing the implementation details of that function. Ensuring the soundness of local reasoning requires eliminating aliases.

- *Preservation of Good Object-Oriented Programming Practices.* A programmer writing in the Clean++ discipline should be guided by the rules and structures of the discipline to write code that makes judicious use of data abstractions to maximize modularity.

- *Familiarity.* A Clean++ program should be familiar to a C++ programmer. It should "look like" C++ and the behavior of the program should not be unexpected to an experienced C++ programmer.

In total, the Clean++ discipline comprises eight rules as well as several informal guidelines. This discussion distinguishes between two kinds of software components: "low-level" components with implementations that fall outside the discipline (because they use vairables of types not in the discipline) and "high-level" components that make exclusive use of Clean++ types.

Mutability The Clean++ discipline has been developed with a specific focus on *mutable* data because immutable data preempts the reasoning problems posed by aliased references. Immutable types in Clean++ have no marked differences compared to their counterparts in idiomatic C++, so they are ignored in this discussion.

6.2.1 Soundness of Local Reasoning

Local reasoning is a technique by which automated verifiers are able to tractably verify the correctness of relatively large programs. There are many language features of mainstream programming languages that complicate local reasoning—and, in some cases, break it entirely.

Perhaps the leading cause for unsound local reasoning is the prevalence of aliased references. When two variables refer to the same piece of memory, the modification of the data through one of the variables will change the value of the other (at least, most people would say so). It is thus desirable for a programming discipline that wishes to maintain the soundness of local reasoning to preclude aliases where possible and advertise or encapsulate them where necessary.

There are several options for maintaining the soundness of local reasoning in Clean++. The first is to prefer statically allocated stack variables, which by default are *copied* on assignment. Of course, the performance of stack variables is prohibitively poor in many cases so the possiblity of dynamically-allocated objects must be accounted for.

Experience has shown that it is possible to develop real-world software with dynamically-allocated objects that has no aliases whatsoever [81], and design patterns that support such components are preferred when applicable. A good way to prevent aliases is to develop types with appropriate move constructors and move assignment operators, and to use them everywhere. An example of such a type from the C++ standard template library is `std::unique_ptr`, a "smart pointer" that expresses singular ownership of its contents. Any assignment of a `std::unique_ptr` must be enclosed in a call to `std::move`, which relinquishes ownership. A program in which all raw pointers are replaced with unique pointers is a program in which there are no aliases.

There are, however, situations that necessitate data sharing of some form—usually aliasing. In such cases, the Clean++ discipline encourages the use of the `std::shared_ptr` type, which *advertises* the fact that a variable might have an alias. Put another way, in a traditional C++ program, the default for any reference is that it may be aliased. To prevent such aliasing, the programmer must do something special such as using `std:unique_ptr`. On the other hand, in a program that follows the Clean++ discipline, the default for any reference is that it may *not* be aliased and the programmer must explicitly allow such aliasing, if needed, with `std::shared_ptr`. The situations in which a shared pointer is necessary are rare and include the implementation of cyclic data structures.

Clean++ Rule 1

All pointers are instances of `std::unique_ptr`. In the rare situations when data sharing is absolutely necessary, `std::shared_ptr` is used.

Another source of unsoundness in local reasoning is null pointers. If a function takes a parameter that might be null, and it attempts to dereference a null pointer, this is a failure of local reasoning. The most obvious way to avoid null dereferences is to prevent null pointers entirely. Experience has shown that eliminating null pointers entirely is, in general, possible, but it leads to code that might be unfamiliar to C++ programmers used to dealing with nullable pointers. In Clean++, pointers are initialized at the point of their declaration to refer to a default object of the appropriate type. This grants to some important reasoning benefits, though it is not always feasible for performance reasons, especially in Clean++ implementations of low-level components (*i.e.*, those that are implemented without other Clean++ components), and for that reason Clean++ permits null pointers in low-level implementations provided they *are never leaked to client code.* Eliminating null references eliminates a significant category of run-time errors, making programs written in Clean++ safer by default.

Clean++ Rule 2

If a null pointer is used in the implementation of a Clean++ component, it is never leaked to client code.

Parameter Passing and Return Types

Every parameter to a method in a Clean++ program should be an rvalue reference. Although this rule causes Clean++ to deviate somewhat from idiomatic C++ more than other rules, the benefits of doing so far outweigh the familiarity concerns.

Clean++ Rule 3

Every method parameter is passed as an rvalue reference.

Rules regarding ownership force Clean++ to account for situations in which, for example, several parameters to a method are to have their values changed. One way to solve this problem is to permit pass-by-reference, although that is avoided because it creates aliases. Since all of the parameters to a method are rvalue references (that is, they are *moved* into the method rather than copied), the arguments lose their values. Sometimes, however, keeping the argument values around is necessary for one reason or another—normally performance. Therefore, rule 3 has a closely-related counterpart, rule 4: `std::tuple` is the default return type of methods in Clean++. Each method should return both any newly-created objects and the updated values of all arguments as components of a tuple. This idiom allows Clean++ to simulate "pass-by-swap" as is implemented in verification- and reasoning-focused languages such as

RESOLVE [128] as well as “in-out” parameters such as in Ada or Swift [26, 7]. A secondary positive side effect of rule 4 is that is easy to automatically translate more traditional-looking method headers and calls into Clean++-conformant headers and calls.

Clean++ Rule 4

Every method has a return type of `std::tuple`, in which the first component(s) are objects created by the method, if any, and the rest are the new values of the arguments in left-to-right order.

Exception: Member Functions

Member functions in C++ always pass their receiver by reference,[13] and because member functions are crucial to producing good C++ code, this language-level limitation on member functions creates an exception to the Clean++ rules above: the receiver to a member function may, indeed, be passed by reference (and its value might be changed).

Exception: Single or Nonexistent Return Values

Sometimes, a method will be simple enough that using a tuple as described in rule 4 would involve a tuple that has a single component (or is empty). In that case, the method need not wrap the return value in a tuple. Situations in which this exception applies include methods for which *every* argument is intended to have its ownership relinquished by the client or in which there is only one parameter (excluding the distinguished parameter). Examples of this exception in practice appear in the

[13]It is possible in C++ to pass a receiver as an rvalue reference with a ref-qualified member function, but the complications that arise when doing so in a way that is compatible with the rest of Clean++ are overwhelming and therefore ref-qualified functions are not recommended in Clean++.

Clean++ `stack` component below in section 6.4.2: neither the `push` nor `pop` method returns a tuple.

6.2.2 Support for Good Object-Oriented Programming Practices

In Clean++ a programmer is guided, by the encapsulation restrictions on aliases and null references, to implement software components with many layers of data abstraction. Doing so allows her to have total control over aliases and to keep them confined to a single abstraction layer. The principle of alias control is, as noted above, one of the cornerstones of the Clean++ discipline. A consequence of this pattern is the following rule.

Clean++ Rule 5

No method introduces an alias that is visible to the client.

A key feature of software components in Clean++ is that they implement an efficient no-argument initializer that produces a coherent value for the type. Doing so ensures that Clean++ can make the guarantee that a client never sees a null pointer, even immediately after the declaration of a new variable.[14]

Clean++ Rule 6

Each component implements a no-argument initializer, which can be made efficient.

Every component in Clean++ must have an efficient and "correct" implementation of `std::move`. This allows a client to leverage built-in C++ move semantics to maintain efficiency without relying on pointer semantics. Correctness in this case

[14]Some components, such as an Array, might use lazy initialization for performance reasons. Null pointers in such situations are not a problem because their existence is totally hidden from the client (see rule 2).

imposes a stronger requirement than the C++ specification. In particular, a moved object in Clean++ must be left in a *consistent initial state* for that type, as if the no-argument initializer had been called. (By contrast, the C++ language specification places no restrictions on the resulting value of a moved object).

Clean++ Rule 7

Each component deletes its copy constructor and copy assignment operator. Instead, it implements a move constructor and move assignment operator.

The software components in the Clean++ library are declared in the `cleanpp` namespace, and every Clean++ software component extends `cleanpp::base`, which is roughly analogous to Java's `Object` (the superclass of all Java types) but limited to Clean++ types.

Listing 6.3: The `cleanpp::base` abstract class.

```
namespace cleanpp {
  class base {
  public:
    base() = default;
    virtual ~base() = default;
    virtual void clear() = 0;
  };
}
```

The `cleanpp::base` class defines one pure virtual method, `clear()`. The purpose of this method is to provide an efficient way to set an object's abstract value to an initial value for its type—that is, one produced by the zero-argument initializer (required by the Clean++ discipline).

Clean++ Rule 8

Each component extends the `cleanpp::base` abstract class, directly or indirectly.

6.2.3 Familiarity

One way in which familiarity is achieved in Clean++ is the manner in which language constructs related to move semantics (*e.g.*, `std::move`) are encapsulated within class implementations as far as possible. This encapsulation means that client code does not include extra syntactic clutter that may be unfamiliar to a C++ programmer. In other words, it is possible for both novice and expert C++ programmers to adopt Clean++. The syntax is simple enough for the former, and familiar enough for the latter. These qualities hold despite a software component built in Clean++ having a somewhat different structure from the idiomatic C++ style.

6.2.4 Additional Considerations Going Beyond the Discipline

The following considerations are not necessarily a novel contribution of Clean++, rather they are widely accepted principles for writing great software. They are discussed here only as examples of well-known software engineering principles that need not be cast aside to realize the reasoning benefits of writing software in the Clean++ discipline.

Abstraction and Reasoning

Components in Clean++, by their modular nature, typically provide a clear distinction between the abstract value and concrete value of a variable. The benefits are similar to the use of "model" variables in [50]. A consequence of using many levels of abstraction to write software in Clean++ is that often the abstract value of a given type will be substantially different from the concrete value for that type. For this reason, most components in Clean++ do not use public fields in classes and instead opt for well-named methods and functions which reflect the *abstract* value of the type and not the *concrete* value. However, this tends to make Clean++ components less familiar to C++ programmers, and making use of public fields does not—necessarily—complicate reasoning. It is therefore permissible to use public fields in Clean++ components, though their use should be considered carefully.

Observability and Controllability

Two design principles in identifying suitable operations for interfaces in Clean++ are *observability* and *controllability*, as motivated by Weide *et al.* to guide the design of abstract data types (ADTs) [142]. The principle of observability states that a client of a component should be able to distinguish between any two unequal values in the ADT's state space using some combination of operations on the ADT. Controllability, on the other hand, says that every value in the ADT's state space is reachable through some combination of operations of the ADT. An approach to designing software components (ADTs) using these principles as a guide tends to produce interfaces that are, in a sense, "sufficient". It is for that reason that Clean++ interfaces should be designed with these principles in mind.

6.3 Prototypical Clean++ Component Structure

The implementations of Clean++ components based on the rules of section 6.2 are only one part (albeit the most complex part) of a Clean++ software component. In most cases, classes of the kind discussed above are not directly used by a client of a component; instead the client declares variables of "flex types" that are wrappers around the implementation types. A flex type in Clean++ has several properties:

1. It is essentially a wrapper around a `std::unique_ptr` to an object of some implementation class.

2. It provides a default implementation type for that object—that is, a client of a Clean++ component with a flex type need not have any knowledge of the implementation(s).

3. It has a class hierarchy that mirrors the structure of the class hierarchy for the implementation types (*e.g.*, if the implementation includes both a kernel and secondary interface, so does the flex type).

4. Every method in the implementation type has a sibling method in the flex type, the implementation of which is simply a redirection to the same method in the implementation.[15]

The flex type pattern closely resembles the "bridge" design pattern, but has several key differences. First, it is not intended to decouple the implementation from the abstraction—such decoupling is done via C++ abstract classes on the implementation side. As a consequence, the abstraction (*i.e.*, the flex type) is a concrete class, not an abstract class. Second, the flex type has an interface that is always *identical* to the interface of the implementation. Of course, the flex type pattern does enjoy several of the benefits of the bridge pattern, most notably the ability to change the implementation at run-time, during the lifetime of a variable.

A Clean++ component family has a structure such as in fig. 6.1. Because a client will use a flex type, Clean++ naming convention dictates that the flex type(s) be named with the canonical name of the component, and that the implementation type(s) be suffixed with "`_impl`".

[15]The exception to this general rule is that when a component is structured with both a kernel and secondary interface, the secondary methods must additionally perform a static cast.

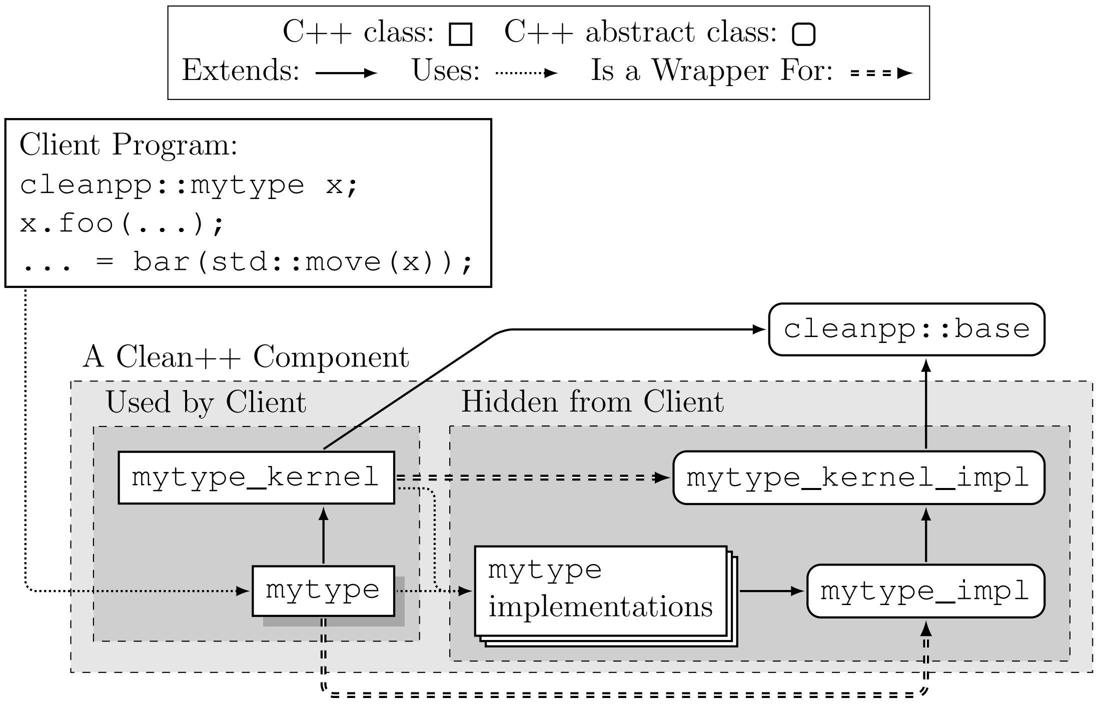


Figure 6.1: Prototypical structure of a Clean++ software component.

The primary purpose of including flex types in the stucture of a Clean++ component is to enable polymorphism without the need for complicated syntax using explicit instances of `std::unique_ptr` or calls to `std::make_unique`. To this end, a flex type provides a special initializer to replace the no-argument initializer of the component's "`_impl`" class that takes one (unused) argument, a variable of the implementation type to be used for this instance of the flex type. Typically, this argument is provided at the point of the initializer call by way of a call to the no-argument constructor of the implementing class. Listings 6.4 and 6.5 show some simple client code and a sample flex type.

Listing 6.4: Client code using `cleanpp::stack`.

```
int main() {
  cleanpp::stack<int> s;
  s.push(4);
  s.push(5);
  s.push(6);
  int x = s.pop();
  std::cout << s; // prints "<5, 4>"
}
```

Listing 6.5: Example flex type for `cleanpp::stack`.

```
namespace cleanpp {

template<typename Item>
class stack: public base {
protected:
  template <typename I>
  using _flex_stack_def_t = linked_stack<I>;
  static_assert(std::is_base_of<stack_impl<int>,
   _flex_stack_def_t<int>>::value, "_flex_stack_def_t␣must␣
   derive␣from␣stack_impl<Item>");


  std::unique_ptr<stack_impl<Item>> rep_;
public:
  stack() : rep_(std::make_unique<_flex_stack_def_t<Item>>()
   ) { }

  template<template<typename> class I>
  stack(__attribute__((unused)) const I<Item>& impl): rep_(
   std::make_unique<I<Item>>()) {
    static_assert(std::is_base_of<stack_impl<Item>, I<Item
   >>::value, "Template␣parameter␣I␣must␣derive␣from␣
   stack_impl<Item>");
  }

  stack(const stack<Item> &o) = delete;
  stack(stack<Item>&& o): rep_(std::move(o.rep_)) {
    o.rep_ = std::make_unique<_flex_stack_def_t<Item>>();
  }
  template<template<typename> class I>
  stack(stack<Item>&& o, __attribute__((unused)) const I<
   Item>& impl): rep_(std::move(o.rep_)) {
    static_assert(std::is_base_of<stack_impl<Item>, I<Item
   >>::value, "Template␣parameter␣I␣must␣derive␣from␣
   stack_impl<Item>");
```

```
    o.rep_ = std::make_unique<I<Item>>();
  }

  stack<Item>& operator=(const stack<Item>& o) = delete;
  stack<Item>& operator=(stack<Item>&& other) {
    if (&other == this) {
      return *this;
    }
    rep_ = std::move(other.rep_);
    other.rep_ = std::make_unique<_flex_stack_def_t<Item>>()
;
    return *this;
  }
  template<template<typename> class I>
  stack<Item>& operator=(stack<Item>&& other, __attribute__
((unused)) const I<Item>& impl) {
    static_assert(std::is_base_of<stack_impl<Item>, I<Item
>>::value, "Template␣parameter␣I␣must␣derive␣from␣
stack_impl<Item>");
    if (&other == this) {
      return *this;
    }
    rep_ = std::move(other.rep_);
    other.rep_ = std::make_unique<I<Item>>();
    return *this;
  }

  void clear() {
    this->rep_->clear();
  }

  virtual void push(Item&& x) {
    rep_->push(std::forward<Item>(x));
  }

  virtual Item pop() {
    return rep_->pop();
  }

  virtual bool is_empty() const {
    return rep_->is_empty();
  }

  bool operator==(stack<Item>& other) {
    return *this->rep_ == *other.rep_;
  }

  friend std::ostream& operator<<(std::ostream& out, stack<
   Item>& o) {
    return out << *o.rep_;
  }
};
}
```

One important feature of the flex type is that it identifies a default implementation (line 7 in listing 6.5). When the flex type is instantiated, information about the implementation type is lost, so when a `stack` (or other flex type) is move-assigned, because rule 2 restricts null references, a new (empty) stack must be initialized in its place and since this requires a class name, a default type is provided. It is important to note that the client need not know about this default type—a client can simply declare a variable of type `stack` without identifying an implementation class (line 2 in listing 6.4). A drawback to the flex type pattern is that a variable of a flex type might have its implementation type changed over the course of its lifetime without the client being made aware of it. Of course, this will not affect the *correctness* of a program but it might have an impact on *performance*. When performance is a concern, the client should be aware of the performance profiles of the implementation options and select appropriately: a programmer always has the *option* of using a particular implementation.

A flex type can be easily generated—automatically, in principle—for Clean++ components that have both "kernel" and "secondary" interfaces such as `mytype` from fig. 6.1, as well as for those that provide just one interface. Implementing a flex counterpart for a secondary interface involves a static cast of the private variable rep_ and while the method bodies in that case have some complex syntax, the client code is exceptionally clean. However, because a flex type such at mytype can be automatically generated from the mytype_impl class, a human need not ever look at the “ugly” code in the flex type class.

6.4 Illustrative Clean++ Component Design and Implementation

Designing abstract data types (or software components) to support sound local reasoning requires some changes to the usual way of programming in C++. To illustrate the ideas, this section considers a relatively straightforward resource example followed by a more detailed stack example.

6.4.1 Implementing a Non-Template Component

As a simple example, we present a class implementation of the type cleanpp::resource that can be used as the type parameter to any of the several collectionn-type classes in the Clean++ library. This example serves two purposes. First, it demonstrates how a non-template type might be implemented in the Clean++ discipline, and second, it provides a context that should be familiar to C++ programmers—one of resource ownership and transfer thereof.

Listing 6.6: Implementation of a Clean++ non-template component.

```
namespace cleanpp {
  class resource_impl: public base {
  private:
    int i_;
  public:
    resource_impl(const resource_impl& other) = delete;
    resource_impl(resource_impl&& other) {
      i_ = other.i_;
      other.clear();
    }

    resource_impl& operator=(const resource_impl& other) =
  delete;
    resource_impl& operator=(resource_impl&& other) {
      if (&other == this) {
        return *this;
      }
      i_ = other.i_;
      other.clear();
      return *this;
    }
    resource_impl() {
      i_ = 0;
    }
    resource_impl(int i) {
      i_ = i;
    }

    void clear() override {
      i_ = 0;
    }

    void mutate(int x) {
      i_ *= x;
    }

    int access() {
      return i_;
    }
  };
}
```

6.4.2 Template Class Implementation with Move Semantics and Unique Pointer

One way a stack can be implemented is using a linked list of raw pointers, although with such an implementation comes the potential for aliases and reasoning complications. With move semantics and `std::unique_ptr`, it is possible to create an efficient linked implementation of a `stack` while avoiding the reasoning pitfalls that are inherent in raw pointer-based software. Using `std::unique_ptr` everywhere that a traditional linked implementation of a stack would use a raw pointer can dramatically reduce the complexity of reasoning required by the traditional implementation and maintain desirable performance characteristics.

However, a move-based stack implementation on its own is not enough to eliminate reasoning difficulties in its client software: the type of item contained within the stack must also be written in the discipline. An example of such a type, `resource`, is discussed above in section 6.4.1.

Consider a simple singly-linked list implementation such as below in listing 6.7. Since it typically requires no pointers connecting nodes to be aliases, the "next" pointer of each node (and the pointer to the first node) can be replaced. Specifically, a smart pointer type introduced alongside move semantics in C++11, `std::unique_ptr`, can be leveraged as a drop-in replacement for pointers in places where a programmer knows *a priori* that there will be no aliasing, and to prevent unwanted aliasing everywhere else. A unique pointer enforces at compile time (by the deletion of the copy constructor and copy assignment operator) that there are no aliases to its contents.[16]

Because the usual behavior of a stack in C++ involves creating aliases to the contents of the stack, the behavior of a stack in Clean++ will necessarily be different than the behavior of the stack in the C++ standard library. Specifically, a call to `top()` on a `std::stack` returns an *alias* to the object at the top of the stack, and

[16]Given the full power of C++, it is technically possible to subvert the guarantee of `std:unique_ptr` through a convoluted series of pointer manipulations. However, this process is sufficiently exotic that it is beyond the scope of ordinary software development and extremely unlikely to happen by accident.

a call to pop() simply discards the object formerly at the top of the stack. This problem can be solved simply: eliminate the top() method altogether and modify the pop() operation to remove *and return* the object at the top of the stack. This behavior change is relatively small, so developers familiar with std::stack will not be totally out of their element working with a Clean++ stack. (Developers familiar with stacks in other languages will immediately be comfortable with this behavior.)

Listing 6.7: Code for a linked implementation of the Clean++ stack.

```
namespace cleanpp {
  template <typename T>
  class stack_impl: public base {
  private:
    class node: base {
    public:
      T contents;
      std::unique_ptr<node> next;

      node(): contents(), next() {}

      node(T&& new_contents):
      contents(), next() {
        std::swap(contents, new_contents);
      }

      node(node const &other) = delete;
      node(node&& other):
      contents(std::move(other.contents)),
      next(std::move(other.next)) {
        other.clear();
      }

      node& operator=(const node& other) = delete;
      node& operator=(node&& other) {
        if (&other == this) {
          return *this;
        }
        contents = std::move(other.contents);
        next = std::move(other.next);
        other.clear();
        return *this;
      }

      void clear() {
        contents = T();
        next.reset();
      }
    };

    std::unique_ptr<node> top_ptr_;
  public:
    stack_impl<T>() { }

    stack_impl<T>(stack_impl<T> const &other) = delete;
    stack_impl<T>(stack_impl<T>&& other):
    top_ptr_(std::move(other.top_ptr_)) {
      other.clear();
    }

    stack_impl<T>& operator=(const stack_impl<T>& other) =
   delete;
    stack_impl<T>& operator=(stack_impl<T>&& other) {
      if (&other == this) {
        return *this;
      }

      top_ptr_ = std::move(other.top_ptr_);
      other.clear();
      return *this;
    }

    void clear() override {
        if (!is_empty()) {
            top_ptr_.reset();
        }
    }

    void push(T&& x) override {
        top_ptr_ = std::make_unique<stack_node>(std::forward
   <T>(x), std::move(top_ptr_));
    }

    T pop() override {
        assert(!is_empty());
        T pop = top_ptr_->contents();
        top_ptr_ = top_ptr_->next();
        return std::move(pop);
    }

    bool isEmpty() const override {
      return top_ptr_ == nullptr;
    }
  };
}
```

The decision to eliminate the `top()` method is important because it enables the elimination of aliases in the Clean++ `stack` component: a client need not first acquire a reference (*i.e.*, an alias) to the item at the top of the stack before removing it. The `push` operation also deviates from `std::stack` in that it takes as an argument an rvalue reference to the item to be placed at the top of the stack. The primary consequence of this decision is that the client no longer owns the object they pass to a call to `push`. In the client program, the argument is surrounded by a call to `std::move` to advertise this fact, and after the operation the value of the argument is an initial value for its type.

6.4.3 Implementing New Components By Reusing Existing Ones

A savvy developer can leverage the reasoning benefits provided by one component by reusing it to easily implement other software components which then exhibit the same nice reasoning properties.

For an example, we consider a reuse of `cleanpp::stack` component from section 6.4.2. Doubly-linked lists are used to great effect as the underlying data structure for abstractions such as a list with a cursor, which describes a series of items in which the client can *insert* a new element at the cursor position, *remove* the element at the cursor position, and *advance* or *retreat* the cursor. Navigating a doubly-linked list in this way is extremely efficient as the data structure provides a natural way of doing so. Unfortunately, by their nature, doubly-linked lists rely on aliases. Since alias avoidance is a key goal of Clean++, it is desirable to implement this data type without any aliases at all.

We can do so using a pair of stacks, one of which represents the list contents preceding the cursor and the other representing the remaining list contents. Advancing or retreating the cursor, then, is done by popping an element from one stack and then pushing it onto the other. Insertion and removal, similarly, are done by pushing an element onto or popping one off of a stack. It was shown above that the push and pop operations on `stack` are efficient. The entire `listwithcursor_impl` class takes about 30 SLOC (see listing 6.8), and exhibits the desired reasoning characteristics of a Clean++ component.

Listing 6.8: An implementation of `listwithcursor_impl` built with a pair of Stacks.

```
namespace cleanpp {
  template <class T>
  class listwithcursor_impl: public base {
  private:
    stack<T> prec_ { };
    stack<T> rem_ { };
  public:
    void advance() {
      T x{ };
      rem_.pop(x);
      prec_.push(x);
    }

    void retreat() {
      T x{ };
      prec_.pop(x);
      rem_.push(x);
    }

    void insert(T& x) {
      prec_.push(x);
    }

    void remove(T& x) {
      prec_.pop(x);
    }

    bool isAtEnd() {
      return rem_.length() == 0;
    }

    bool isAtFront() {
      return prec_.length() == 0;
    }
  };
}
```

6.4.4 Implementing Components with Unavoidable Aliasing

Some work has been done recently with respect to verifying behavioral correctness of programs with aliased references [91, 92, 134]. The products of this research are somewhat exotic and require language-level primitives, so until their results are adopted by C++, an alternative is needed in cases where aliases are unavoidable. Such a case is a linked-list implementation of a Queue, with a pointer to both the head of the list (marking the front of the queue) and the last node in the list (marking the tail of the queue). The tail pointer is an alias to the `next` pointer of the second-to-last node in the list. Eliminating the tail pointer solves the aliasing problem, but incurs unacceptable performance penalties. Such a queue can be implemented within the Clean++ discipline because *no aliases will ever leak to the client*, and thus the soundness of modular reasoning is preserved. Listing 6.9 shows such an implementation.

Listing 6.9: A linked implementation of a queue in the Clean++ discipline.

```cpp
namespace cleanpp {
  template <typename T>
  class queue: public base {
  private:
    class node: base {
    private:
    public:
      T contents;
      std::shared_ptr<node> next;
      node(): contents(), next() {}

      node(T&& new_contents):
      contents(std::move(new_contents)), next() {}

      node(node const &other) = delete;
      node(node&& other):
      contents(std::move(other.contents)),
      next(std::move(other.next)) {
        other.clear();
      }

      node& operator=(const node& other) = delete;
      node& operator=(node&& other) {
        if (&other == this) {
          return *this;
        }
        contents = std::move(other.contents);
        next = std::move(other.next);
        other.clear();
        return *this;
      }

      void clear() {
        contents = T();
        next.reset();
      }
    };
    std::shared_ptr<node> top_ptr_;
    std::shared_ptr<node> tail_ptr_;
  public:
    queue<T>(): top_ptr_(), tail_ptr_() { }

    queue<T>(queue<T> const &other) = delete;
    queue<T>(queue<T>&& other):
    top_ptr_(std::move(other.top_ptr_)),
    tail_ptr_(std::move(other.tail_ptr_)) {
      other.clear();
    }

    queue<T>& operator=(const queue<T>& other) = delete;
    queue<T>& operator=(queue<T>&& other) {
      if (&other == this) {
        return *this;
      }

      top_ptr_ = std::move(other.top_ptr_);
      tail_ptr_ = std::move(other.tail_ptr_);
      other.clear();
      return *this;
    }

    void clear() {
      top_ptr_.reset();
      tail_ptr_.reset();
    }

    void enqueue(T& x) {
      auto new_tail = std::make_shared<node>(std::move(x));
      if (tail_ptr_ != nullptr) {
        tail_ptr_->next = new_tail;
      } else {
        top_ptr_ = new_tail;
      }
      // Alias!!
      tail_ptr_ = new_tail;
    }

    void dequeue(T& x) {
      std::swap(x, top_ptr_->contents);
      std::swap(top_ptr_, top_ptr_->next);
    }
```

```
82
83      bool isEmpty() const {
84        return top_ptr_ == nullptr;
85      }
86    };
87  }
```

The similarities to the linked implementation of `cleanpp::stack` are significant, with the obvious difference that `std::shared_ptr` is used in places where `std::unique_ptr` was used in listing 6.7. This difference is because of the alias introduced at line 75 that is a consequence of maintaining a pointer to the tail of the queue in addition to the head. Importantly, this alias is never leaked to the client (nor is the null reference indicating the end of the queue). A shared pointer is used rather than a raw pointer for several reasons: first, to advertise the fact that this variable might have aliases at any given time, and second to let the C++ compiler manage memory for the programmer.

6.5 Evaluating Clean++

The Clean++ discipline was presented to an undergraduate audience in order to evaluate its efficacy and usability in two kinds of programs: “client” software that make use of existing Clean++ components and “implementation” software that comprise the totality of a Clean++ component. Qualitative evaluation showed that the syntactic burden is not overwhelming; it it easy for an undergraduate student to successfully incorporate syntax related to move semantics into both kinds of programs. However, it was clear that leveraging an existing C++ compiler to raise errors when the discipline is violated was crucial to efficient development, which justified the attention paid to such features from the beginning.

Feedback from evaluation of the discipline informed several descisions in its refinement. To give one example, at several points during evaluation it was deemed necessary to write a function that both changed the value of one of its arguments and returned a separate object. However, because move semantics was a key part of Clean++ from the beginning, this led to the inclusion of rule 4, by which every method has a return type of `std::tuple`.

6.5.1 Concurrency

It is possible to achieve a degree of fork-join parallelism within the Clean++ discipline. However, because arguments are passed by move in Clean++, parallel operations that ought to share an argument are in fact not able to do so—with one exception: member functions, to which the receiver is passed by reference. An object could, for example, serve as the receiver to several parallel member function calls, although the programmer must take care not to share any other arguments between the parallel calls. Reasoning about such a program could then involve logical machinery such as the A/P Calculus introduced in chapter 3 and be specified using a construct such as the non-interference contract from chapter 5. Other concurrency tools such as threads, semaphores, and locks were not considered in the development or evaluation of the discipline.

6.6 Clean++ vs. Other RESOLVE Disciplines

Clean++ is not the first attempt to ease reasoning in an industrial language using RESOLVE principles. Over the several-decades-long lifetime of the RESOLVE project [129], there have been a number of projects that bridge the gap between purely academic and purely practical software engineering, including several disciplines similar in principle to Clean++. However, in each of the previous efforts, mapping to and from RESOLVE was hindered by the ubiquity of reference semantics and aliases in the target language, which in most cases forced compromises by the designers. This work, in contrast, leverages existing features of C++ (namely, move semantics) to curtail aliases *without* inventing new language features or completely redefining existing ones.

6.6.1 RESOLVE/Ada

In his dissertation [80], Hollingsworth describes the RESOLVE/Ada discipline for building high-quality reusable software components in the Ada programming language. He identifies several key properties of programs in the discipline that overlap broadly with those of Clean++. They are:

- Correctness
- Composability
- Reusability
- Understandability

The RESOLVE/Ada discipline is intended to serve as a guide for Ada programmers wishing to bring the principles of RESOLVE to Ada, including formal specifications, a clear distinction between abstract and concrete state, and the idea that a software component might be composed of *several* correct implementations (not just one). The properties of composability and reusability are closely aligned with properties of Clean++ programs in that satisfying these properties in a meaninfgul way requires the programmer to tightly control rogue references and prevent implementation details from leaking to the client. Reference control is achieved in Clean++ through the use of `std::unique_ptr` and in RESOLVE/Ada through two "principles" (which correspond to Clean++ "rules") prohibiting repeated arguments to procedure calls and prohibiting arguments to procedure calls that are part of global state accessible by the procedure.

6.6.2 RESOLVE/C++

Developed approximately contemporaneously with RESOLVE/Ada is the RESOLVE/C++ discipline [82], which includes not only a set of principles for developing C++ programs that adhere to the RESOLVE ideology, but also a set of macros and redefined C++ operators to support such development. As RESOLVE/C++ was developed in the mid-1990s—well before move semantics was introduced to the C++ standard or even proposed—it was limited by the features of C++ at the time to manage aliases (which were essentially nonexistent). The workaround in the discipline was two-fold: redefine the "`&=`" operator to be "swap" and prohibit assignment and copy construction entirely by declaring the relevant functions as "private". These decisions make RESOLVE/C++ programs entirely unlike idiomatic C++ programs, limiting its utility as a general-purpose software engineering discipline.

Clean++ takes these two principles a step further by leveraging relatively recent advancements to C++ (including move semantics) by *deleting* copy constructors and assignment operators entirely and replacing them with their moving counterparts. Despite its shortcomings as a general-purpose software discipline, RESOLVE/C++

proved to be a useful precursor to Clean++ as it justified the practicality and feasibility of the flex type pattern in Clean++ in which the implementation type is provided as a template parameter.

6.6.3 RESOLVE and Java

Relatively more recent work by members of the RESOLVE group in bridging the academic/practical divide has been focused on Java [83, 145].

The first foray by Hollingsworth, *et al.*, produced preliminary "Rules of Engagement" (*i.e.*, a discipline) for Java that codified the preference of *transfer* as the primary data movement operation. Although swapping has long been at the core of the RESOLVE project, it is not practical in Java.[17] In the Rules of Engagement for Java, non-aliasing data movement is implemented through the explicit use of null references and by restricting the use of the assignment operator. The various rules ensure that there is only ever one (non-temporary) reference to each object. This is accomplished, in part, by simulating ownership transfer by explicitly assinging the value **`null`** to variables that have relinquished ownership.

Zaccai, in his dissertation [145], formalized a discipline for writing Java programs in which it is sound to reason about objects in terms of their abstract or mathematical value and, in general, ignore concerns about references and the heap. The so-called Resolve Java discipline prefers ownership transfer as the primary data movement

[17]Because every type (except the primitive types) in Java is a reference type and Java does not allow a programmer to access the reference directly, there are two ways to implement swapping—neither of which are acceptable. First, an extra level of indirection could be added to each component; that is, the implementation of a component would have a single field: a reference to an object that contains the "real" data. Under this *modus operandum*, the implementation of swap would involve the usual three-line assignment sequence to swap the inner references of two variables. If there is no extra indirection, the implementation of swap would require language features that do not exist in Java, such as pass-by-reference, multiple return values, or clever use of a preprocessor.

operation, and implements it for mutable types with the `transferFrom` operation. Assignment (by reference) and copying are still permitted in Resolve Java, but they are avoided. In pursuit of the stated goal of identifying a subset of Java that is amenable to modular verification, several Java software components are specified and verified. Zacai additionally formally specified a variety of ubiquitous Java statements such as `assert` statements, branching (**`if`**/**`else`**) statements, and iteration (**`for`** /**`while`**) statements, including **`break`** and **`continue`**. The Resolve Java discipline has been used as a tool for teaching undergraduate computer science students sound software engineering principles [10].

6.7 Clean++ vs. Rust

Rust addresses many of the problems identified and addressed by Clean++. Consequentially, it is natural to ask whether Clean++ has already been made redundant by Rust. The answer is no. First, while Rust is an entirely new language that introduces complex new ideas[18] and requires programmers to learn new syntax, Clean++ identifies a subset of a well-established programming language that virtually all programmers are familiar with.

Additionally, while Rust provides limited facilities for true object-oriented programming, C++ provides a full suite of capabilities for building robust, modular, object-oriented programs that make use of abstract data types, polymorphism, and inheritance. These capabilities are drawn upon in Clean++ to maintain client/implementer separation and to encourage the use of many levels of abstraction—things

[18]The ownership system alone has been the topic of discussion for several weekly meetings of a group of university professors and graduate students . . . and it still is not completely understood by all of them.

that would, in Rust, be at best difficult and cumbersome to achieve. For comparison's sake, appendix A contains a sample component written both in Clean++ and Rust.

Finally, Rust's solutions to many of the problems addressed in this paper are framed in terms of memory management and memory safety—*not* ease of reasoning. This does not mean that Rust's fixes do not improve reasoning (they do), but it does mean that in some cases Rust may not go far enough toward easy reasoning to make it amenable to automated formal verification (although there is work in that direction [27]). We believe that Clean++ does, since it is inspired directly by verification-focused languages such as RESOLVE. By including comment-based formal contracts, it is believed that programs written from the ground up in Clean++ could be formally verified to be correct.

6.8 Repository of Clean++ Components

A small library of components written in Clean++ has been developed to accomplish a variety of programming tasks. It is available in a public GitHub repository [141].

6.9 Conclusions

While the dangers posed by aliased references are well-understood, there is relatively little work on curtailing them in mainstream programming languages, and especially little work on doing so with the stated goal of easing reasoning. A variety of longstanding efforts by members of the RESOLVE community have produced disciplines with such a goal, but none were able to leverage language-level support for ownership transfer in C++ (simply because they were developed too early). The

Clean++ discipline, therefore, advances the state of the art in two respects. Programs written in Clean++ do not have aliases, which by intention eases reasoning about them, and the discipline leverages C++ move semantics to provide ownership transfer as the primary data movement operation without altering the language.

Chapter 7: Conclusions and Future Directions

For all their advantages, reusable data abstractions are not widely used in parallel programs. This research has shown that principles of modularity and abstraction can be leveraged to ease development and facilitate modular verification of parallel software.

7.1 Logical Framework and Proof of Soundness

In pursuit of enabling modular reasoning in the presence of concurrency, a calculus for effects was introduced that abstracts an object and operations on that object. The effects were used as the basis for a programming model in which many layers of abstraction is the norm and parallel programs are deterministic by default.

The A/P Calculus presented is currently suitable as the foundation for any specification construct that describes objects that are either statically partitioned (as they are in the non-interference contracts of chapter 5) or dynamically partitioned (as they would be in the specification of, *e.g.*, a list abstraction).

The semantics of the proposed programming model were discussed and shown to generalize earlier results about the semantics of parallel programs. Specifically, the definition of "non-interference" was relaxed from prior work, enabling the semantics of parallel programs to be well-defined even when objects are shared among processes.

The utility of these semantics was demonstrated by reasoning about a simple parallel program involving a concurrent queue.

7.2 Clean Semantics and Decomposable Data Abstractions for Fork-Join Parallelism

Even without machinery such as the A/P Calculus, there are several classes of parallel programs that can be made amenable to modular verification given enough language support. This support might come in the form of clean semantics, which guarantees in part that there are no aliases. A methodology which leverages clean semantics for developing data abstractions that can be used in parallel programs without novel specification constructs was presented, and a family of components built using that methodology was evaluated. The methodology guides the development of abstract data types that can be decomposed before a parallel section and recomposed afterward. Such decomposition enables a verifier or programmer to reason about a parallel section of code as if it were sequential because no two parallel statements modify the same variable and are therefore non-interfering.

7.3 Abstract Non-Interference Specifications

Considered component design alone is not always a practical way to enable safe parallelism, so a new specification construct, the non-interference contract, was introduced that can enable modular reasoning about parallelism even in situations that would, with other verification techniques, force the breakdown of the principles of modular reasoning and data abstraction. A non-interference contract is an implementation of the programming model based on the A/P Calculus in the RESOLVE programming language, and is accompanied by a **`cobegin`** statement. A client reasons about parallel operation calls using the non-interference contract's effects clauses, which describe in abstract terms how the operation calls interact with the object's state. Importantly, a non-interference contract does not reveal any implementation details and the partition structure is completely modular. It was used to specify the parallelism capabilities of a bank account and a concurrent bounded queue and to reason about a queue implementation using one of the decomposable components introduced in chapter 4.

7.4 Writing C++ Programs That Are Easy To Reason About

Finally, a discipline for programming in C++ was presented that enables sound local reasoning about a wide variety of sequential software. Components were developed in the discipline that exhibit a variety of properties that ease reasoning, such as alias freedom. Despite the perception of move semantics as a purely performance-improving trick, this work shows that there is significant potential for C++ move semantics to be broadly applied in large software projects not only to improve performance but also to simplify reasoning and, thus, increase their robustness. It is also shown that even without specialized analysis tools, clean semantics (and the reasoning simplifications that it entails) can be achieved in C++ through judicious—and careful—use of move semantics and `std::unique_ptr`.

7.5 Future Directions

Although fork-join parallel programs are an important subset of parallel programs, they represent a tiny fraction of the universe of parallel software. There are important directions of inquiry that stem from this work to further expand the reach of modular reasoning.

7.5.1 A/P Calculus

The current iteration of the A/P Caluclus does not support the modeling of synchronization. However, a model of synchronization might be achieved by adding to each effect a third set that represents pieces that are "atomically affected" by an operation. This change in turn could impact the calculus itself or the programming model for which it serves as the basis.

7.5.2 Leveraging Clean Semantics for Fork-Join Parallelism

The limitations of the methodology for designing decomposable components are directly related to the limitations of RESOLVE and the A/P Calculus. While a "shared" implementation of the various array abstractions in chapter 4 is discussed at a high level, the actual code is conspicuously absent. There is not, at present, a mechanism in the RESOLVE language to handle objects that are decomposed at the abstract level but still share data at the implementation level. Enabling such an implementation is not a goal of this work but it is a natural extension of the work by Sun in his recent dissertation [132].

7.5.3 Abstract Non-Interference Specifications

At present, a non-interference contract is capable of identifying a static number of pieces for an object's partition. Improving the expressivity of the partition syntax is a key goal and would enable the specification of the parallelism capabilities of a variety of collection types including arrays, sets, and dictionaries, and enable the implementation of more sophisticated program statements such as parallel iteration. Expanding the reach of the A/P calculus would have a cascading effect on the utility of non-interference contracts. For example, if a model of synchronization is achieved in the A/P Calculus, it could be readily realized through non-interference contracts.

There is not, at present, a capability to automatically verify the "respects" relation, but there is a road map to achieve such automation. There is also no automation involved in verifying the correctness of client programs with **`cobegin`** statements; however, because reasoning about a non-interfering parallel program is equivalent to reasoning about a sequential program with the same statements, a rudimentary implementation of the **`cobegin`** statement in RESOLVE would be trivial.

7.5.4 Clean++

The Clean++ discipline, as presented here, is rigid enough in many respects that a static analyzer could be developed to check automatically that code adheres to several of the core ideas, although such a validator has not been implemented. There are several aspects of the discipline that would be relatively easy to check and others that would prove more challenging. Specifically, it would be trivial to check for any raw pointers and warn the programmer if any are found. If a smart pointer other than `std::unique_ptr` or `std::shared_ptr` is found, a warning could also be dispatched suggesting that the programmer employ one of those two "acceptable" types. Ensuring that all user-defined types delete the copy constructor and assignment operator and have replaced them with a move constructor and move assignment operator would not be difficult (although showing their correctnesss is another problem entirely). Finally, many instances of null references or alias “leakage” to the client could be caught with only a moderately sophisticated static analysis.

Challenges to automating Clean++ validation include checking the correctness of the no-argument constructor and not just its existence. Options for enabling this kind of sophisticated analysis include employing comment-based specifications regarding abstraction and initial values. In most cases, default initializers are straightforward and determining the abstract value generated by such a method, given an appropriate specification, could be achieved.

7.5.5 Technology Transfer

RESOLVE has been used both as a language and a set of principles to teach computer science to undergraduate students, with projects of that kind reaching back several decades at multiple institutions. The non-interference contract (and the accopanying **`cobegin`** statement) could be implemented in RESOLVE to enable the teaching of concurrency and parallel software development in those courses. Enabling the teaching of parallel programming contemporaneously with general formal reasoning principles could improve the quality of parallel and concurrent software written by undergraduate students, and in turn the quality of software they produce after they graduate.

Moreover, the benefits afforded by the Clean++ discipline make it a good candidate to be a tool for teaching formal reasoning to computer science students. In Clean++, a programmer explicitly advertises the moving behavior of components and forces the client to use rvalue references for arguments to methods. Thus, she never needs to reason about the possibility of aliases: every object always has a unique owner. This simplification is a boon to students, especially to those who are only beginning to learn about programming.

Another benefit of the strictness of the Clean++ discipline with regard to the ease of reasoning is that some C++ code that introduces aliases are flagged as erroneous by any off-the-shelf C++ compiler as long as the programmer is using Clean++ components. This feature could help beginning students see how to write programs that avoid aliases, and to develop an understanding of the mechanisms of data transfer that are used in many of today's popular programming languages.

The ideas proposed by this dissertation are directly applicable to academic programming languages such as RESOLVE. Chapter 6 describes a discipline in C++ to apply various principles of RESOLVE to a mainstream language, and it could serve as a framework to further the reach of these ideas. For example, as discussed near the end of that chapter, a degree of parallelism is achievable in the Clean++ discipline without any modification. Clean++ programs also posess properties that make them amenable to local and modular reasoning, so it is feasible that C++ could be used as a compilation target for RESOLVE programs that have been verified to be correct—including RESOLVE programs that make use of a new **`cobegin`** construct.

7.6 Conclusion

In conclusion, this work has expanded the reach of modular reasoning to include a class of fork-join parallel programs, provided a sound foundation for such reasoning, and demonstrated how to write easy-to-reason-about software in C++. It has additionally built scaffolding that could enable modular reasoning about complex parallel and concurrent programs, and identified important directions of inquiry to that end.

Bibliography

[1] ACSL: ANSI/ISO C specification language, version 1.11. https://frama-c.com/download/acsl_1.11.pdf.

[2] Altran UK. http://www.altran.co.uk/.

[3] Frama-C software analyzers. http://frama-c.com/.

[4] PVS specification and verification system. http://pvs.csl.sri.com/.

[5] SPARKPro. http://www.adacore.com/sparkpro/.

[6] Spec#. https://www.microsoft.com/en-us/research/project/spec/.

[7] Subprograms - learn.adacore.com. https://learn.adacore.com/courses/intro-to-ada/chapters/subprograms.html#parameter-modes.

[8] VeriFast. https://people.cs.kuleuven.be/{\~{}}bart.jacobs/verifast/.

[9] Class DPJPartition<type T, region R>. http://dpj.cs.illinois.edu/DPJ/Download_files/DPJRuntime/DPJPartition.html, 2010.

[10] Software I and II Course Sequence. http://web.cse.ohio-state.edu/software/web/index.html, 2012.

[11] Parallel Patterns Library (PPL). https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl, 2016.

[12] Google C++ style guide. https://google.github.io/styleguide/cppguide.html, 2018.

[13] Learn c++. https://learncpp.com, 2018.

[14] readonly (C# Reference). `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly`, 2020.

[15] Array.prototype.slice(). `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice`, 2021.

[16] ArraySlice. `https://developer.apple.com/documentation/swift/arrayslice`, 2021.

[17] Built-in Functions. `https://docs.python.org/3/library/functions.html?highlight=slice#slice`, 2021.

[18] C Struct Declaration. `https://en.cppreference.com/w/c/language/struct`, 2021.

[19] Class: Array. `https://ruby-doc.org/core-3.0.0/Array.html#method-i-slice`, 2021.

[20] PHP: array_slice. `https://www.php.net/manual/en/function.array-slice.php`, 2021.

[21] std::list. `https://en.cppreference.com/w/cpp/container/list`, 2021.

[22] Structure types (C# Reference). `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct`, 2021.

[23] Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, mar 2006.

[24] Bruce Adcock, Paolo Bucci, Wayne D. Heym, Joseph E. Hollingsworth, Timothy Long, and Bruce W. Weide. Which pointer errors do students make? *SIGCSE 2007: 38th SIGCSE Technical Symposium on Computer Science Education*, pages 9–13, 2007.

[25] Bruce M Adcock. *Working Towards the Verified Software Process.* PhD thesis, Ohio State University, 2010.

[26] Apple Inc. *The Swift Programming Language.* Swift Programming Series. Apple Inc., 2018.

[27] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[28] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[29] Anindya Banerjee, David A Naumann, and Stan Rosenberg. Regional Logic for Local Reasoning About Global Invariants. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 387–411, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[31] Mike Barnett, Bor-Yuh Chang, Robert DeLine, Bart Jacobs, and K Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin / Heidelberg, 2006.

[32] Mike Barnett, Robert DeLine, Manuel Fähndrich, K Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs*, 3(6):27–56, jun 2004.

[33] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[34] Bernhard Beckert, Martin Giese, Reiner Hähnle, Vladimir Klebanov, Philipp Rümmer, Steffen Schlager, and Peter H Schmitt. The KeY System 1.0 (Deduction Component). In F Pfenning, editor, *Proceedings, International Conference on Automated Deduction, Bremen, Germany*, LNCS 4603. Springer, 2007.

[35] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007.

[36] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. DoubleChecker: Efficient sound and precise atomicity checking. *ACM SIGPLAN Notices*, 49(6):28–39, 2014.

[37] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, software-only region conflict exceptions. *ACM SIGPLAN Notices*, 50(10):241–259, 2015.

[38] François Bobot and Jean-Christophe Filliâtre. Separation Predicates: A Taste of Separation Logic in First-Order Logic. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin Heidelberg, 2012.

[39] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wroc\l{}aw, Poland, aug 2011.

[40] Robert L Bocchino Jr., Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[41] Matko Botinčan, Matthew Parkinson, and Wolfram Schulte. Separation Logic Verification of C Programs with an SMT Solver. *Electron. Notes Theor. Comput. Sci.*, 254:5–23, oct 2009.

[42] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using First-Order Theorem Provers in the Jahob Data Structure Verification System. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin / Heidelberg, 2007.

[43] Stephen Brookes. A Semantics for Concurrent Separation Logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 16–34, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[44] P. Bucci, T. J. Long, and B. W. Weide. Do we really teach abstraction? *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, pages 26–30, 2001.

[45] Paolo Bucci, Joseph E. Hollingsworth, Joan Krone, and Bruce W. Weide. Part III: implementing components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):40–51, 1994.

[46] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In Orna Grumberg, editor,

Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings, pages 400–411. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[47] Tevfik Bultan and Constance Heitmeyer. Applying infinite state model checking and other analysis techniques to tabular requirements specifications of safety-critical systems. *Design Automation for Embedded Systems*, 12(1):97–137, 2008.

[48] Patrice Chalin, Joseph R Kiniry, Gary T Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 342–363, Berlin, Heidelberg, 2006. Springer-Verlag.

[49] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.

[50] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35(6):583–599, 2005.

[51] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-order Imperative Programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 79–90, New York, NY, USA, 2009. ACM.

[52] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Procee*, pages 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[53] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[54] Michelle Cook, Megan Fowler, Jason O Hallstrom, Joseph E Hollingsworth, Tim Schwab, Yu-Shan Sun, Murali Sitaraman, Germán Andrés Delbianco,

Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, Jean-Christophe Filliâtre, Claude Marché, Megan Fowler, Michelle Cook, Jason O Hallstrom, Joseph E Hollingsworth, Kevin Plis, Tim Schwab, Yu-Shan Sun, Murali Sitaraman, Colin S Gordon, Michael D Ernst, Dan Grossman, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, Joe Duffy, Wayne Heym, Paolo Sivilotti, Joseph E Hollingsworth, Joan Krone, Murali Sitaraman, Nigamanth Sridhar, Nabil M Kabbani, Joan Krone, Murali Sitaraman, Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Dirk Leinenbach, Wolfgang Paul, Elena Petrova, Xavier Leroy, Kalyan C Regula, Hampton Smith, Heather Harton Keown, Jason O Hallstrom, Nigamanth Sridhar, Murali Sitaraman, Yu-Shan Sun, Murali Sitaraman, and Aditi Tagore. seL4: Formal Verification of an OS Kernel. In Werner Damm and Holger Hermanns, editors, *CoRR*, volume 52 of *SEFM '05*, pages 433–448, New York, NY, USA, may 2016. Clemson University - School of Computing, ACM.

[55] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S P\uas\uareanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[56] Nicholas Coughlin and Graeme Smith. Rely/Guarantee Reasoning for Noninterference in Non-Blocking Algorithms. In *Proceedings - IEEE Computer Security Foundations Symposium*, volume 2020-June, 2020.

[57] David Crocker and Judith Carlton. Verification of C Programs Using Automated Reasoning. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, pages 7–14, Washington, DC, USA, 2007. IEEE Computer Society.

[58] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[59] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Mechanically Verified Proof Obligations for Linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1), jan 2011.

[60] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-Guarantee Reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] Matthew B Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S Păsăreanu, Hongjun Zheng, and Willem Visser. Tool-supported Program Abstraction for Finite-state Verification. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 177–187, Washington, DC, USA, 2001. IEEE Computer Society.

[62] Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part II: specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29–39, 1994.

[63] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[64] Xinyu Feng. Local Rely-guarantee Reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[65] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*, pages 173–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[66] Cormac Flanagan and Stephen N Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[67] Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 338–349, New York, NY, USA, 2003. ACM.

[68] Robert W Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[69] Stephen N Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[70] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 42, pages 483–496. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, aug 2015.

[71] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-compatible persistent transactions. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–74, 2020.

[72] Colin S Gordon, Michael D Ernst, Dan Grossman, and Matthew Parkinson. Verifying Invariants of Lock-free Data Structures with Rely-Guarantee and Refinement Types. *ACM Transactions on Programming Languages and Systems*, 39(3):11:1—-11:54, may 2017.

[73] Douglas E Harms and Bruce W Weide. Copying and Swapping: Influences on the Design of Reusable Software Components. *IEEE Trans. Softw. Eng.*, 17(5):424–435, may 1991.

[74] John Harrison. Formal proof - theory and practice. In *Notices of the AMS*, pages 1395–1406, 2008.

[75] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[76] John Hatcliff, Gary T Leavens, K Rustan M Leino, Peter Müller, and Matthew Parkinson. Behavioral Interface Specification Languages. *ACM Comput. Surv.*, 44(3):16:1—-16:58, jun 2012.

[77] Maurice P Herlihy and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[78] Howard E. Hinnant, Peter Dimov, and Dave Abrahams. A proposal to add move semantics support to the C++ language. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm`, 2002.

[79] C A R Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, oct 1969.

[80] Joseph E Hollingsworth. *Software component design-for-reuse: a language-independent discipline applied to ADA.* PhD thesis, 1992.

[81] Joseph E Hollingsworth, Lori Blankenship, and Bruce W Weide. Experience Report: Using RESOLVE/C++ for Commercial Software. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 11–19, New York, NY, USA, 2000. ACM.

[82] Joseph E. Hollingsworth, Sethu Sreerama, Bruce W. Weide, and Sergey Zhupanov. Part IV: RESOLVE components in Ada and C++. *ACM SIGSOFT Software Engineering Notes*, 19(4):52–63, 1994.

[83] Joseph E. Hollingsworth and Bruce W. Weide. Some Preliminary Rules of Engagement for Java. In *Proceedings of Resolve Workshop 2006*, Virginia Tech, Blacksburg, VA, 2006.

[84] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Proceedings of the 7th International Workshop on Specification and Verification of Component-Based Systems*, SAVCBS'08, pages 83–88, 2008.

[85] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[86] C B Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, oct 1983.

[87] C B Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, mar 1996.

[88] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic Parallelism via Liquid Effects. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 45–54, New York, NY, USA, 2012. ACM.

[89] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, San Francisco, CA, USA, 2018.

[90] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K Rustan M Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience Report. In Michael Butler and Wolfram Schulte,

editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 154–168. Springer Berlin Heidelberg, 2011.

[91] Gregory Kulczycki. *Direct Reasoning.* Phd dissertation, Clemson University, School of Computing, 2004.

[92] Gregory Kulczycki, Murali Sitaraman, Bruce W Weide, and Atanas Rountev. A Specification-based Approach to Reasoning About Pointers. In *Proceedings of the 2005 Conference on Specification and Verification of Component-based Systems*, SAVCBS '05, New York, NY, USA, 2005. ACM.

[93] Gregory Kulczycki, Hampton Smith, Heather Harton, Murali Sitaraman, William F Ogden, and Joseph E Hollingsworth. The Location Linking Concept: A Basis for Verification of Code Using Pointers. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 34–49. Springer Berlin Heidelberg, 2012.

[94] Viktor Kuncak and Martin Rinard. An Overview of the Jahob Analysis System: Project Goals and Current Status. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, page 285, Washington, DC, USA, 2006. IEEE Computer Society.

[95] Gary T Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R Smith, and Aaron Stump. Roadmap for Enhanced Languages and Methods to Aid Verification. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 221–236, New York, NY, USA, 2006. ACM.

[96] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, oct 1998.

[97] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, may 2006.

[98] Gary T Leavens, K Rustan M Leino, and Peter Müller. Specification and Verification Challenges for Sequential Object-oriented Programs. *Form. Asp. Comput.*, 19(2):159–189, jun 2007.

[99] K Rustan M Leino. Specification and verification of object-oriented software. In Manfred Broy, Wassiou Sitou, and Tony Hoare, editors, *Engineering Methods and Tools for Software Safety and Security*, Volume 22 NATO Science for Peace

and Security Series - D: Information and Communication Security, pages 231–266. IOS Press, 2009.

[100] K Rustan M Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[101] K Rustan M Leino and Valentin Wüstholz. The Dafny Integrated Development Environment. In *F-IDE*, 2014.

[102] K.Rustan M Leino and Rajit Manohar. Joining specification statements. *Theoretical Computer Science*, 216(1):375–394, 1999.

[103] Timothy J Long, Bruce W Weide, Paolo Bucci, David S Gibson, Joseph E Hollingsworth, Murali Sitaraman, and Stephen H Edwards. Providing intellectual focus to CS1/CS2. In John Lewis, Jane Prey, Daniel Joyce, and John Impagliazzo, editors, *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, 1998, Atlanta, Georgia, USA, February 26 - March 1, 1998*, pages 252–256. ACM, 1998.

[104] Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-Guarantee Protocols. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*, pages 334–359. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[105] J Misra and K M Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, jul 1981.

[106] Carroll Morgan. The Specification Statement. *ACM Trans. Program. Lang. Syst.*, 10:403–419, 1988.

[107] Michal Moskal, Thomas Santen, and Wolfram Schulte. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674, pages 23–42. Springer, January 2009.

[108] Mayur Naik and Alex Aiken. Conditional Must Not Aliasing for Static Race Detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 327–338, New York, NY, USA, 2007. ACM.

[109] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[110] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: the RESOLVE framework and discipline. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, 1994.

[111] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[112] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, 2001. Springer-Verlag.

[113] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 268–280, New York, NY, USA, 2004. ACM.

[114] Oracle. Package java.util.concurrent. `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html`, 2021.

[115] Susan Owicki and David Gries. An Axiomatic Proof Technique for Parallel Programs. In David Gries, editor, *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, pages 130–152. Springer New York, New York, NY, 1978.

[116] S Owre, J M Rushby, and and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[117] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic Using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 773–789, Berlin, Heidelberg, 2013. Springer-Verlag.

[118] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic with Trees and Data. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 711–728, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[119] Eli Pozniansky and Assaf Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM.

[120] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, mar 2007.

[121] John C Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[122] Jake Roemer, Kaan Genç, and Michael D. Bond. SmartTrack: Efficient predictive race detection. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 747–762, 2020.

[123] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.

[124] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. Legato: End-to-end bounded region serializability using commodity hardware transactional memory. *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 1–13, 2017.

[125] Vineet Singh, Iulian Neamtiu, and Rajiv Gupta. Proving Concurrent Data Structures Linearizable. In *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pages 230–240, 2016.

[126] Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M Friedman, Heather Harton, Wayne Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith, and Bruce W Weide. Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing*, 23(5):607–626, 2011.

[127] Murali Sitaraman, Timothy J Long, Bruce W Weide, E James Harpner, and Liqing Wang. A Formal Approach to Component-based Software Engineering: Education and Evaluation. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 601–609, Washington, DC, USA, 2001. IEEE Computer Society.

[128] Murali Sitaraman and Bruce Weide. Component-based Software Using RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, oct 1994.

[129] Murali Sitaraman and Bruce W Weide. Special Feature: Component-Based Software Using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–22, 1994.

[130] Neelam Soundarajan, Raffi Khatchadourian, and Johan Dovland. Reasoning about the behavior of aspect-oriented programs. *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications, SEA 2007*, 1:198–202, 2007.

[131] Aaron Stump. *Verified Functional Programming in Agda.* Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.

[132] Yu-Shan Sun. *Towards Automated Verification of Object-Based Software with Reference Behavior.* Phd thesis, Clemson University, School of Computing, 2018.

[133] Yu-Shan Sun, Joan Krone, and Murali Sitaraman. Automated Verification With and Without Reference Behavior. Technical Report RSRG-17-04, Clemson University - School of Computing, Clemson, SC 29634, nov 2017.

[134] Yu-Shan Sun, Bruce W Weide, Diego Zaccai, Paolo A G Sivilotti, and Murali Sitaraman. The RESOLVE Approach for Achieving Modular Verification: Progress and Challenges in Addressing Aliasing. Technical Report RSRG-17-03, Clemson University - School of Computing, Clemson, SC 29634, oct 2017.

[135] S Tucker Taft. ParaSail: A Pointer-Free Pervasively-Parallel Language for Irregular Computations. *The Art, Science, and Engineering of Programming*, Volume 3(Issue 3), 2019.

[136] Julian Tschannen, Carlo A Furia, Martin Nordio, and Bertrand Meyer. Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods*, pages 382–398, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[137] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, jul 2008.

[138] Viktor Vafeiadis and Matthew Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In Luís Caires and Vasco T Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory: 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007. Proceedings*, pages 256–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[139] Alan Weide, Paolo A G Sivilotti, and Murali Sitaraman. Enabling Modular Verification with Abstract Interference Specifications for a Concurrent Queue. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories,*

Tools, and Experiments, pages 119–128, Cham, 2016. Springer International Publishing.

[140] Alan Weide, Paolo A G Sivilotti, and Murali Sitaraman. An Array Abstraction to Amortize Reasoning About Parallel Client Code. In *Computing Conference*, London, UK, 2021. Springer.

[141] Alan Weide, Paolo A.G. Sivilotti, Murali Sitaraman, and William Janning. Clean++ Component Library. `https://github.com/alanweide/cleanpplib`, 2021.

[142] B W Weide, S H Edwards, W D Heym, T J Long, and W F Ogden. Characterizing observability and controllability of software components. In *Proceedings of Fourth IEEE International Conference on Software Reuse*, pages 62–71, 1996.

[143] Daniel Welch. *Scaling Up Automated Verification: A Case Study and a Formalization IDE for Building High Integrity Software.* PhD thesis, Clemson University, School of Computing, 2019.

[144] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. Abstract Specifications for Concurrent Maps. In Hongseok Yang, editor, *Proceedings of the 26th European Symposium on Programming (ESOP'17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 964–990. Springer, apr 2017.

[145] Diego Sebastian Zaccai. *A Balanced Verification Effort for the Java Language.* PhD thesis, Ohio State University, 2016.

[146] Yannick Zakowski, David Cachera, Delphine Demange, Gustavo Petri, David Pichardie, Suresh Jagannathan, and Jan Vitek. Verifying a Concurrent Garbage Collector with a Rely-Guarantee Methodology. *Journal of Automated Reasoning*, 63(2), 2019.

[147] Karen Zee, Viktor Kuncak, and Martin Rinard. Full Functional Verification of Linked Data Structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 349–361, New York, NY, USA, 2008. ACM.

[148] Karen Zee, Viktor Kuncak, and Martin C Rinard. An Integrated Proof Language for Imperative Programs. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 338–351, New York, NY, USA, 2009. ACM.

Appendix A: Object-Oriented Programming in Rust vs. Clean++

Consider a simple mutable type, called *Natural* that is modeled by the natural numbers ℕ and is unbounded (this is similar to a "Big Integer" found in libraries for many languages). Our implementation, however, will be represented by a single **`unsigned long`** (or **`u64`** in Rust) for simplicity's sake, so it is bounded in reality. Keep in mind while reading that an alternative implementation (*e.g.*, based on a Stack) could be employed to make it effectively unbounded.

Listing A.1: Implementation in Rust of a Natural type.

```
pub mod natural {
    use std::fmt;

    pub trait Kernel: fmt::Debug + fmt::Display {

        /*
         type NATURAL is integer
            exemplar n
            constraint n >= 0
            initialization ensures n = 0

         natural_number_kernel is modeled by NATURAL
         */

        /*
         updates  self
         requires 0 <= d and d < 10
         ensures  self = #self * 10 + d
         */
        fn multiply_by_radix(&mut self, digit: u64);

        /*
         updates  self
         ensures  #self = self * 10 + d and
                  0 <= d and d < 10
         */
        fn divide_by_radix(&mut self, digit: &mut u64);

        /*
         ensures is_zero = (self = 0)
         */
        fn is_zero(&self) -> bool;

        /*
         ensures self = 0
         */
        fn clear(&mut self);
    }

    pub trait Secondary: Kernel
    {
        /*
         updates self
         ensures self = #self + 1
         */
        fn increment(&mut self) {
            let mut last_digit = 0;
            self.divide_by_radix(&mut last_digit);
            last_digit += 1;
            if last_digit == 10 {
                last_digit -= 10;
                self.increment();
            }
            self.multiply_by_radix(last_digit);
        }

        /*
         updates  self
         requires self > 0
         ensures  self = #self - 1
         */
        fn decrement(&mut self) {
            let mut last_digit = 0;
            self.divide_by_radix(&mut last_digit);
            if last_digit == 0 {
                last_digit += 10;
                self.decrement();
            }
            last_digit -= 1;
            self.multiply_by_radix(last_digit);
        }

        /*
         replaces self
         ensures  self = n
         */
        fn set_from_u64(&mut self, mut n: u64) {
            self.clear();
            if n > 0 {
                let d = n % 10;
                n /= 10;
                self.set_from_u64(n);
                self.multiply_by_radix(d);
            }
        }
    }

    #[derive(Debug)]
    pub struct Bounded {
        n: u64,
    }

    impl Bounded {
        pub fn new() -> Bounded {
            Bounded { n: 0 }
        }
   }

    impl fmt::Display for Bounded {
        fn fmt(&self, f: &mut fmt::Formatter) ->
          fmt::Result {
            write!(f, "{}", self.n)
        }
    }

    impl Kernel for Bounded {
        fn multiply_by_radix(&mut self, digit: u64) {
            self.n *= 10;
            self.n += digit;
        }
        fn divide_by_radix(&mut self, digit: &mut u64) {
            *digit = self.n % 10;
            self.n /= 10;
        }
        fn is_zero(&self) -> bool {
            self.n == 0
        }
        fn clear(&mut self) {
            self.n = 0;
        }
    }

    impl Secondary for Bounded {}

    /*
     updates lhs
     ensures lhs = #lhs + rhs
     */
    pub fn add(
      lhs: &mut std::boxed::Box::<impl Secondary>,
      mut rhs: std::boxed::Box::<impl Secondary>) {
        let mut lhs_last = 0;
        let mut rhs_last = 0;
        lhs.divide_by_radix(&mut lhs_last);
        rhs.divide_by_radix(&mut rhs_last);
        lhs_last += rhs_last;
        if lhs_last > 10 {
            lhs_last -= 10;
            lhs.increment();
        }
        if !rhs.is_zero() {
            add(lhs, rhs);
        }
        lhs.multiply_by_radix(lhs_last);
    }
}

```

```
fn main() {
    use natural::Secondary;
    let mut n =
      std::boxed::Box::new(natural::Bounded::new());
    n.set_from_u64(42);
    let mut m =
      std::boxed::Box::new(natural::Bounded::new());
    m.set_from_u64(21);
    natural::add(&mut n, m);
    println!("n␣=␣{}", n);
}
```

Listing A.2: Implementation in Clean++ of a Nautral type, with **`#include`** directives and **`namespace`** declarations elided.

```
/**********************\
 * natural_number.hpp *
\**********************/

class natural_number_kernel: public clean_base {
    /*
     type NATURAL is integer
        exemplar   n
        constraint n >= 0
        initialization ensures n = 0

     natural_number_kernel is modeled by NATURAL
     */
public:
    static const int RADIX = 10;

    /*
     ensures is_zero = (this = 0)
     */
    virtual bool is_zero() const = 0;

    /*
     updates  this
     requires 0 <= d and d < RADIX
     ensures  this = #this * RADIX + d
     */
    virtual void multiply_by_radix(int d) = 0;

    /*
     updates  this
     ensures  #this = this * RADIX + d and
              0 <= d and d < RADIX
     */
    virtual void divide_by_radix(int &d) = 0;

    /*
     ensures '==' = (this = other)
     */
    bool operator==(natural_number_kernel &other);

    friend std::ostream& operator<<(std::ostream& out,
   natural_number_kernel& o);
};

class natural_number_secondary:
  public natural_number_kernel {
public:
    /*
     updates this
     ensures this = #this + 1
     */
    virtual void increment();

    /*
     updates  this
     requires this > 0
     ensures  this = #this - 1
     */
    virtual void decrement();

    /*
     replaces this
     requires n >= 0
     ensures  this = n
     */
    virtual void set_from_long(long n);

    /*
     updates x
     ensures x = #x + y
     */
    friend void add(
      std::unique_ptr<natural_number_secondary> &x,
      std::unique_ptr<natural_number_secondary> &y);

    /*
     updates  x
     requires x >= y
     ensures  x = #x - y
     */
    friend void subtract(
      std::unique_ptr<natural_number_secondary> &x,
      std::unique_ptr<natural_number_secondary> &y);
```

```
};

/**********************\
 * natural_number.cpp *
\**********************/

// natural_number_kernel
bool natural_number_kernel::operator==(
  natural_number_kernel &other) {
    bool ans = false;
    if (other.is_zero() && this->is_zero()) {
        ans = true;
    } else if (other.is_zero() == this->is_zero()) {
        int last_this, last_other;
        this->divide_by_radix(last_this);
        other.divide_by_radix(last_other);
        if (last_this == last_other) {
            ans = *this == other;
        }
        this->multiply_by_radix(last_this);
        other.multiply_by_radix(last_other);
    }
    return ans;
}

std::ostream& operator<<(
  std::ostream& out,
  natural_number_kernel& o) {
    if (o.is_zero()) {
        out << 0;
    } else {
        int d;
        o.divide_by_radix(d);
        if (!o.is_zero()) {
            out << o;
        }
        out << d;
        o.multiply_by_radix(d);
    }
    return out;
}

// natural_number_secondary
```

```
void natural_number_secondary::increment() {
    int d = 0;
    divide_by_radix(d);
    d++;
    if (d == RADIX) {
        d -= RADIX;
        increment();
    }
    multiply_by_radix(d);
}

void natural_number_secondary::decrement() {
    assert(!is_zero());
    int d = 0;
    divide_by_radix(d);
    d--;
    if (d < 0) {
        d += RADIX;
        decrement();
    }
    multiply_by_radix(d);
}

void natural_number_secondary::set_from_long(long n) {
    assert(n >= 0);
    if (n == 0) {
        clear();
    } else {
        long nLeft = n / RADIX;
        set_from_long(nLeft);
        multiply_by_radix(n % RADIX);
    }
}

void add(
  std::unique_ptr<natural_number_secondary> &x,
  std::unique_ptr<natural_number_secondary> &y) {
    int x_low;
    x->divide_by_radix(x_low);
    int y_low;
    y->divide_by_radix(y_low);
    if (!y->is_zero()) {
        add(x, y);
    }
    x_low += y_low;
    if (x_low >= natural_number_secondary::RADIX) {
        x_low -= natural_number_secondary::RADIX;
        x->increment();
    }
    x->multiply_by_radix(x_low);
    y->multiply_by_radix(y_low);
}

void subtract(
  std::unique_ptr<natural_number_secondary> &x,
  std::unique_ptr<natural_number_secondary> &y) {
    int x_low;
    x->divide_by_radix(x_low);
    int y_low;
    y->divide_by_radix(y_low);
    if (!y->is_zero()) {
        subtract(x, y);
    }
    x_low -= y_low;
    if (x_low < 0) {
        x_low += natural_number_secondary::RADIX;
        x->decrement();
    }
    x->multiply_by_radix(x_low);
    y->multiply_by_radix(y_low);
}

/*****************\
 * bounded_nn.hpp *
\*****************/

class bounded_nn: public natural_number_secondary {
public:
    bounded_nn(long n = 0);

    bounded_nn(bounded_nn const &other) = delete;
    bounded_nn(bounded_nn&& other);

    bounded_nn& operator=(const bounded_nn& other) =
      delete;
    bounded_nn& operator=(bounded_nn&& other);

    bool operator==(const bounded_nn &other);

    void clear() override;
    bool is_zero() const override;
    void multiply_by_radix(int d) override;
    void divide_by_radix(int &d) override;
    void increment() override;
    void decrement() override;
    void set_from_long(long n) override;

    friend std::ostream& operator<<(
      std::ostream& out,
      bounded_nn& o) {
        return out << o.n_;
    }
private:
    long n_;
};

/*****************\
 * bounded_nn.cpp *
\*****************/

bounded_nn::bounded_nn(long n): n_(n) {};

bounded_nn::bounded_nn(bounded_nn&& other):
  n_(std::move(other.n_)) {
    other.clear();
}

bounded_nn& bounded_nn::operator=(bounded_nn&& other) {
    if (&other == this) {
        return *this;
    }

    n_ = other.n_;
    other.clear();
    return *this;
}

bool bounded_nn::operator==(const bounded_nn &other) {
    return this->n_ == other.n_;
```

```
}

void bounded_nn::clear() {
    n_ = 0;
}
bool bounded_nn::is_zero() const {
    return n_ == 0;
}
void bounded_nn::multiply_by_radix(int d) {
    n_ *= RADIX;
    n_ += d;
}
void bounded_nn::divide_by_radix(int &d) {
    d = n_ % RADIX;
    n_ /= RADIX;
}

void bounded_nn::increment() {
    this->n_++;
}

void bounded_nn::decrement() {
    this->n_--;
}

void bounded_nn::set_from_long(long n) {
    this->n_ = n;
}

/************\
 * main.cpp *
\************/

int main() {
    std::unique_ptr<natural_number> n =
      std::make_unique<bounded_nn>();
    n->set_from_long(42);
    std::unique_ptr<natural_number> m =
      std::make_unique<bounded_nn>();
    m->set_from_long(21);
    add(n, m);
    printf("n␣=␣%s", to_str(n));
}
```

When either of the main programs in listings A.1 or A.2 is run, the output is `"n = 63"`, just as is expected from a quick read of the `main` method in either case. The key takeaway from this example is in Rust's type inference. The type of `n` (and `m`, for that matter) in Listing A.1 is `std::boxed::`**`Box`**`::<natural::Bounded>`, while the (explicitly defined) type of the variables in Listing A.2 is `std::unique_ptr <natural_number>`. From a reasoning point of view, the Clean++ version is superior because the *specification* of the behavior of the Rust variables lives in the traits `natural::Kernel` and `natural::Secondary` (respectively, the classes `natural_number_kernel` and `natural_number_secondary` in the Clean++ version), and not in `Bounded`, the implementation.