PROGRAMMING SAFELY WITH WEAK (AND STRONG) CONSISTENCY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

MATTHEW PIERSON MILANO

August 2020

PROGRAMMING SAFELY WITH WEAK (AND STRONG) CONSISTENCY

MATTHEW PIERSON MILANO

Cornell University, 2020

Writing programs against weak consistency is inherently difficult. This dissertation makes the job of writing safe programs against weak consistency easier, by introducing programming languages in which strong guarantees are defended from weakly-consistent influence, and in which programmers can write consistent-by-construction programs atop underlying weakly-consistent replication.

The first of these languages is MixT, a new language for writing *mixed-consistency transactions*. These atomic transactions can operate against data at multiple consistency levels simultaneously, and are equipped with an information-flow type system which guarantees weakly-consistent observations cannot influence strongly-consistent actions.

While mixed-consistency transactions can defend strong data from weak observations, they cannot ensure that fully-weak code is itself correct. To address this, we leverage monotonic data types to introduce a core language of datalog-like predicates and triggers. In this language, programmers can write monotonic functions over a set of monotonic shared objects, ultimately resulting in a boolean. These monotonic, boolean-returning functions are stable predicates: once they have become

true, they remain true for all time. Actions which are predicated on these stable predicates cannot be invalidated by missed or future updates.

This monotonic language sits at the core of Derecho, a new system for building strongly-consistent distributed systems via replicated state machines. Derecho's Shared State Table (SST) implements monotonic datatypes atop Remote Direct Memory Access (RDMA), resulting in a high-performance, asynchronous substrate on which to build Derecho's monotonic language. Using this SST, we have rephrased the Paxos delivery condition monotonically, granting strong consistency despite the underlying asynchronous replication.

Finally Gallifrey exposes the monotonic reasoning properties of Derecho's core language directly to the user, safely integrating monotonic datatypes into a traditional Java-like programming language. Gallifrey allows any object to be asynchronously replicated via *Restrictions* to its interface, allowing only those operations which are safe to call concurrently. Datatypes shared under these restrictions can be viewed monotonically, using a language of predicates and triggers similar to that at the core of Derecho. A novel *linear region*-based type system enforces that shared object restrictions are respected.

BIOGRAPHICAL SKETCH

Matthew Milano was born in New York, New York in 1990. He obtained his Bachelor of Science in Computer Science and Music from Brown University in 2013 and his Masters of Science in Computer Science from Cornell University in 2017. He is a recipient of the 2012 Rose Rosengard Subotnik prize, the 2013 Brown University Senior Prize in Computer Science, an honorable mention for the 2015 National Science Foundation Graduate Research Fellowship, and is a 2015 National Defense Science and Engineering Graduate Fellow. He has been recognized for his service to the Cornell Computer Science Department and his outreach efforts in the community. In his spare time Matthew is an avid reader, board games enthusiast, and academically-trained composer.

ACKNOWLEDGMENTS

The kindness and generosity of the people who have surrounded me during my PhD never ceases to amaze. Without the community surrounding me, attaining a PhD would be nearly impossible. While my research and academic life has been touched by innumerable hands, I would like to single out a few specific individuals who have had an outsized impact on my life and work in graduate school.

I first must call out my advisor, Andrew Myers, and my former advisor, Nate Foster. It is hard to overstate the influence your advisor has on you, and I have been blessed to work under two faculty who are dedicated to making sure that I grow into the researcher they always knew (no matter how I doubted) that I could be. Andrew Myers in particular takes special care to let me pick my own directions and follow my own problems, even if it's not in a direction that he would naturally have taken—and even if it's not clear that success is in easy reach. And he's been right there with me every step of the way, ever ready with a guiding hand in case I stumble.

Tom Magrino and Isaac Sheff were my original mentors when I arrived at the department, and soon grew into my friends and collaborators. Tom's expertise in coordination-avoiding systems was instrumental in understanding Gallifrey's systems challenges, while Isaac's continued quiet assistance has proven invaluable into the final months of my PhD.

Rolph Recto, my closest student collaborator and intellectual sparring partner, always pushed me to justify my designs, to consider every avenue and not just go with what seems most natural to me. Key ideas in Gallifrey (chapter 3)—such as the ability to transition between restrictions—are the result of his insistence. The accessibility of Gallifrey has no doubt improved dramatically as a result.

Ken Birman has been an incredibly impactful mentor, collaborator, and all-but-advisor since the early years of my PhD. His continued insistence that we shouldn't settle just for what seems possible has been the driving force uncountable breakthroughs in Derecho, and his tireless advocacy for his students and projects is legendary. I am forever grateful that he chose to include me.

Fabian Muehlboeck has been an endless source of inspiration, assistance, and companionship for the duration of my PhD. We spent many late nights together, working through each other's research problems, playing games, or staying sane. Every idea in this thesis owes something to Fabian, whether it be late-night debugging help or careful assistance with formal material.

But the largest thanks must be saved for two primary undergraduate research assistants, Patrick LaFontaine and Danny Yang, without whom this dissertation would surely remain incomplete. Patrick took implementation lead in the systems aspects of Gallifrey (chapter 3), integrating shared objects into Antidote's existing CRDT patterns. Danny is responsible for almost the entire Gallifrey compiler, and continued his contributions for months after his graduation. It is my sincere hope that I see both of them as PhD students some day.

This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a, and by NSF grants 1717554 and 1704788

CONTENTS

LIST OF FIGURES

LIST OF TABLES

0 INTRODUCTION

0.1 SETTING THE STAGE

Our interconnected world is increasingly reliant on large, complex distributed programs. They power our homes, mediate our interactions, provide us food, give us jobs, and make our modern life possible. Every day, the scale of these programs—and the volumes of data they must process—grows, leaving distributed programming platforms that appeared future-proof even just a few years ago struggling to cope.

This reality has posed a challenge to distributed systems programming, to put it mildly. In the face of the demands of ever-increasing scale, replication has become essential. Data kept on a single machine is vulnerable to loss if that machine fails, and accesses to that data must contend with high load if the data is popular, or high latency if data is located far from where it is needed. Therefore, in any large-scale distributed system, data must be replicated; copies of that data must exist on multiple machines, allowing the system to place replicas close to where the data is needed, and to ensure some source of the data remains available even when replicas fail.

The trouble with replication is consistency. When we teach programmers how to write concurrent or distributed code, we tell them to imagine that each thread of execution will be arbitrarily interleaved—that the next instruction, or next transaction, will be individually selected and

executed in isolation. Implicit in this model is the idea of memory as a monolith; memory (or a database) enforces sequential access, allowing only a single instruction (or transaction) to execute at a time. This abstraction is powerful, and sits at the root of familiar "synchronous" concurrency control mechanisms such as locks.

Under replication there is no easy monolith. Performant replicated systems must allow all replicas to process operations, threatening the abstraction of a single monolithic monitor. But giving up on sequentiality comes at a high price, invalidating patterns that have served us well for decades.

To avoid paying this price, traditional systems present an interface to memory (or a database) which guarantees strong consistency properties like *linearizability* [Herlihy and Wing, 1988]: that all operations will appear to occur in a total order, consistent with the real-world order in which they were submitted. But the *consensus protocols* [Pease, Shostak, and L. Lamport, 1980] used to enforce these strong consistency properties are expensive and latency-sensitive, limiting the scale of strongly consistent replicated systems and leaving them, for the most part, locked within a single datacenter.

The expense of traditional consensus has led many in the systems community to give up on strong consistency, leading to the rise of a new class of distributed systems. These systems abandon the old, familiar abstractions of ACID database transactions and consistent memory for a new class of nimble, *weakly consistent* distributed systems. These systems promise unprecedented scale, robustness under load and persistent operation even during partial system failures [Brewer, 2010].

There's only one problem, though: these systems are incredibly hard to program against. The hallmark property that all weakly consistent systems share is *asynchronous replication*: the idea that a client performing a write at one replica will have their operation acknowledged before knowledge of it has reached other replicas. This technique can result in seemingly-impossible program traces, shattering our abstraction of memory as a monolith. Under weak consistency, concurrent threads simply do not appear to execute in an interleaved, instruction-at-a-time fashion. Rather, some instructions appear to "teleport" back in time, observing a state of memory which no longer exists.

And that's just the well-behaved weakly consistent systems. Certain memory models in use today—such as the one encoded in the semantics of C11—have even even stranger semantics. These models allow speculative reads that may be justified by some *future* write, leading to a phenomenon of *out-of-thin-air* or *self-justifying* reads in which a thread reads an arbitrary value and then proceeds to write it, producing the very operation that justified their read in the first place [Batty et al., 2011].

Pervasive and familiar patterns possible with strong consistency—such as the simple idea of holding a lock—are simply impossible under weak consistency [Leslie Lamport, Perl, and Weihl, 2000]. In their place, programmers must rely on careful engineering requiring detailed knowledge of the precise consistency guarantees provided by their underlying systems. This can be done; much of the modern web is built on weak consistency. But it is all too easy to get wrong.

0.2 AN OVERVIEW OF CONTRIBUTIONS

The goal of this work is to make it easier to write safe programs against weak consistency, by introducing programming languages in which strong guarantees are defended from weakly consistent influence, and in which programmers can write consistent-by-construction programs atop underlying weakly consistent replication.

We need to start by recognizing that the core challenge of weak consistency—understanding how to write correct programs given a set of confusing primitives—is not just a systems problem. It's a language problem. As this dissertation demonstrates, programming languages give us the tools we need to use consistency safely. Through language design in particular, we can build languages in which *every* program uses consistency safely.

We find ourselves at a crossroads. We can accept the reality of weakly consistent systems, and invest our energy in building languages that ease the task of programming against today's weakly consistent systems. We could revisit the role of consistent replication, engineering new consistent systems—and new ways of using them—better-suited to the modern scale. Or we could change our abstractions, avoiding the need to reason directly about consistency at all.

This dissertation explores each of these directions. Recognizing that the world's data cannot easily be moved, chapter 1 starts by improving the safety of existing weakly consistent systems through a novel abstraction of *mixed-consistency* transactions in the MixT programming language. Next, chapter 2 demonstrates that strong consistency *can* scale by introducing a

new programming language in which consistent-by-construction programs can be built atop asynchronous replication. This language lies at the core of Derecho, a new system for programming with strongly consistent replicated actors. Finally, chapter 3 introduces Gallifrey, a new language for building externally-consistent applications atop an abstraction of weakly consistent replicated actors. To support the static guarantees of Gallifrey, chapter 4 presents a new type system to enforce *reservation safety*, a form of memory-safety property powerful enough to capture freedom from destructive races.

0.3 MEETING THE MIXED-CONSISTENCY WORLD WITH MIXT

0.3.1 *Where We Are Today*

Before we can imagine a world of tomorrow, we must face the world of today: a mishmash of applications and databases offering varying guarantees, with legacy stores[1] straining against their loads and newer weakly consistent systems slowly replacing them. Modern distributed application development is an endless parade of growing pains; while the narrative in academia may feature a new class of systems rising to replace the old, the reality in industry is less "replace" and more "join." While newer, faster systems come online every day to serve emerging applications, legacy systems persist. These legacy systems are large, and complex, and expensive to replace, storing petabytes (at least) and serving live applications. Industry is not in a rush to migrate this data to a new, untested solution after every SOSP.

1 Or data stores, or databases; we will use these terms largely interchangeably.

Industrial distributed systems programmers face this difficult reality: the data that their distributed applications need is rarely stored in just one database [Brown, 2017]. And in our brave new world of weakly consistent stores, with every new database comes a new consistency model, leaving distributed application programmers with a dizzying array of distinct consistency guarantees.

While evidence from industry suggests programming against a single weak consistency model is viable ([Carlson, 2013; Gentz et al., 2017; Klophaus, 2010; Lakshman and Malik, 2010; Plugge, Membrey, and Hawkins, 2010]), albeit with effort, programming against *multiple* such models at once currently is not; beyond the simple cognitive task of understanding each model's guarantees in isolation, programmers need to reason about how those models compose.

In fact, sometimes the composition of two systems fails to guarantee the behaviors that both systems offer individually. Consider linearizability and causal+ consistency as an example. As defined by Herlihy and Wing, *linearizability* guarantees that all operations will appear to occur in the order they were submitted, consistent with the global "wall clock" time of the observers [Herlihy and Wing, 1988]. As defined by Lloyd, Freedman, and Kaminsky, Causal+ consistency offers a weaker guarantee: that an operation will be ordered after any operations which may have caused it, here defined as those operations visible to the issuer at the time of issuance [Lloyd et al., 2011]. It should be intuitively clear that linearizability is strictly stronger than causal+ consistency; we should therefore expect an application which relies on only causal+ and linearizable data stores to witness at least causal+ consistency.

In this we would be mistaken. The trouble is that our question of how to compose distinct consistency models has arisen from a need to write programs which access multiple *mutually-unaware* systems. These systems have no means of exporting their guarantees to one another. If a client issues one operation against a causal store, and a subsequent operation against a linearizable store, it is possible—indeed likely—that the speed of replication for the linearizable operation will outstrip that of the causal one. A subsequent reader at a different replica would then be able to observe the linearizable operation's effect before the causal operation reaches this replica, violating causality.

To make matters worse, programmers are left to fight against this complexity without their most valuable tool: transactions. ACID transactions ([Gray, 1981; Haerder and Reuter, 1983]) are the core concurrency control tool available in database programming; they allow a fragment of code to appear to run in isolation, as though it was accomplished in its entirety with a single operation at the database's consistency level.[2] While transactions are ubiquitous even in weakly consistent systems, it is nearly impossible to build transactions which operate against *multiple* such systems in the same transaction.

Against this backdrop runs our distributed programming ecosystem. Applications, with state spanning multiple stores, built atop guarantees that are hard to understand individually and harder still to understand in composition. All without access to transactions, the primary way to make distributed programming sane.

2 In fact only the the atomic and isolated properties of ACID transactions are necessary here—the AI of ACID. The CD, which stand for consistent and durable, are a relic of disk-based relational databases and have a fluid interpretation in modern systems. It is worth noting that "consistent" here is with respect to relational constraints, and has nothing to do with consistency models.

0.3.2 *Static Information Flow for Consistency*

The first step towards taming this wilderness comes in the form of an observation. When programmers choose which database should store which data, they are implicitly tying the consistency of that database to the invariants they wish to maintain over that data. This makes consistency, rather than being a property of a database or an operation, a property of the *data itself*. When a programmer requests a linearizable transaction to manipulate data, what they are saying is that the guarantees of linearizability—that objects are always up-to-date—are essential to maintaining the invariants on their data. Thus the important thing here is not so much the protocol for reading and writing, but the guarantee of freshness for the data being read and written.

Leveraging this observation forms a correctness condition: if consistency is a property of data, then it should be an error to allow less-consistent observations to influence more-consistent data. This condition effectively makes consistency the analog of integrity; less consistent data is less trustworthy, and should not be allowed to influence more-trustworthy, or more-consistent data.

And while the field of programming languages may be relatively newly-arrived to the problem of programming against multiple levels of distributed consistency, it has made great strides in the problem of programming against a decentralized model of heterogeneous integrity [Askarov and A. Myers, 2010; Biba, 1977; Sabelfeld and A. C. Myers, 2003]. Those solutions apply here; in particular, a standard type system for static integrity information flow can be applied directly to consistency. All one

needs are labels—here the consistency models supported by underlying databases—drawn from a lattice. Such a lattice can be found in the work of Viotti and Vukolić, who have already partially ordered consistency models by the guarantees they provide [Viotti and Vukolić, 2016].

This information-flow type system enforces exactly the property motivated earlier: that it should be an error to allow less-consistent observations to influence more-consistent data. But consistency is not quite integrity. In some ways this is good—consistency can often be recovered simply by waiting for a period of quiescence, or performing a read with a larger quorum of replicas, both much cheaper operations than the sort of multi-participant join required to recover integrity. But consistency is also weaker than integrity in one critical way: the moment an operation has completed, the guarantees provided by consistency begin to weaken. A linearizable read may have returned the most up-to-date value available at the time of the read, but during the time taken to process that value it could well become stale.

Simply put, data atrophies. Because of this, it would be a mistake to simply write programs in an integrity information-flow type system; while the system will ensure that weakly consistent reads do not influence strongly consistent writes, it will be unable to ward against once-consistent reads that, due to their use outside a transaction, no longer represent consistent data.

0.3.3 *Mixed-Consistency Transactions*

The right place to put an information-flow type system for consistency is within the transactions themselves. Through their ACID properties, transactions ensure that the data read within a transaction retains its consistency guarantees for the transaction's duration. But here lies a problem: our target domain features applications which program against *multiple* databases, and there are no efficient and correct transactions mechanisms that span databases. And even for cases which require only a single store, traditional transactions always execute at a *single* consistency level; we'd be stuck paying the performance penalty of the maximum consistency in the transaction, despite the fact that access to less-consistent data is safe under weak consistency. We need a transaction mechanism that can span stores and contain operations at many consistency levels simultaneously; without just upgrading them to the strongest.

This dissertation contains such a mechanism. Chapter 1 of this dissertation presents MixT, a domain-specific language for programming *mixed-consistency* transactions against multiple, mutually-unaware backing stores. MixT leverages the analogy with integrity to implement an information-flow type system for reasoning about consistency within transactions. But it also strengthens the system slightly in the process; rather than simply banning weak-consistency observations from influencing strong-consistency writes, as would be done in a traditional integrity information-flow system, MixT bans weak-consistency observations from influencing strong-consistency *reads* as well. The choice of what data to read *cannot depend* on an observation made at a weaker consistency level.

This strengthening of the type system is not necessary for correctness, but it *does* allow MixT to effectively pipeline all its operations; as no possible weak observation can influence a strong operation, it is possible to execute the strong portion of the transaction in its entirety before proceeding to the weaker portions. Using this insight, MixT implements *cross-store* transactions by chopping a mixed-consistency transaction up into phases, with one phase per consistency level. MixT then runs each component transaction at its corresponding store, using a lightweight runtime mechanism to synchronize these single-store component transactions.

0.3.4 *Problems Still Persist*

MixT successfully reduces the difficulty of programming against multiple consistency models simultaneously back to the difficulty of programming against each of these models in isolation. But this is still a hard problem! MixT's guarantees are only as good at the labels it uses, and it says nothing about whether code within a *single* consistency model is using that model correctly. It is still just as possible to write subtly-incorrect weakly consistent programs with MixT as it was to write them against a single weakly consistent store.

0.4 IN DEFENSE OF STRONG CONSISTENCY WITH DERECHO

There are two ways to address the challenge of programming directly against weak consistency: build new languages in which programs which operate against weak consistency are consistent by construction, or build

new strongly consistent systems which are fast enough to avoid the need for weak consistency in the first place. Neither is a replacement for the other: consistent-by-construction languages are (by necessity) less expressive than general-purpose programming atop strong consistency, while enforcing weak consistency will always introduce less overhead than enforcing strong.

The key is not to view these solutions in isolation, but rather as two components of a single solution. We can build a consistent-by-construction programming language expressly for the purpose of writing a next-generation strongly-consistent distributed system. The resulting system inherits the asynchronous, streaming-style of computation native to weak consistency, without opening itself up directly to the dangers of weak consistency.

0.4.1 *Existing Solutions*

In fact, several consistent-by-construction languages suitable for this task already exist. An example of such a language is Bloom^L, introduced by Peter Alvaro, Neil Conway, and Joe Hellerstein. Bloom^L dramatically reimagines the role of the programming language in distributed systems, stripping down the available operations into a core set of Datalog-like operators guaranteed to be convergent, and consistent, under asynchronous replication [Conway et al., 2012]. Using Bloom^L, programmers can write correct-by-construction distributed code which witnesses the same behaviors under weak consistency as it does under strong, eliminating the role of consistent replication almost entirely in the process. This language is (necessarily) quite limited, but is expressive enough for some applications.

But simply writing a system in the asynchronous style enabled by Bloom^L will not be sufficient to displace weak consistency in the datacenter. Weak consistency, fundamentally, requires consensus; and consensus requires round trips.

Existing work shows us that within a datacenter, the costs of round-trip communication can be managed—especially if emerging network fabrics are in the mix. The domain of emerging hardware continues to open exciting opportunities to reduce the cost of replication, featuring such innovative solutions as using software-defined network switches to push consensus protocols into the network itself [Dang et al., 2015; Jialin Li et al., 2016], or leveraging new Remote Direct Memory Access (RDMA) hardware to bypass not just the CPU, but the copy-heavy design of traditional POSIX sockets entirely [Balakrishnan et al., 2012; Dragojević et al., 2014]. It is this latter technology that has been most directly applied to the issue of replication, with major cloud providers, including Microsoft, rolling out user-accessible RDMA in their public clouds.

Along with this availability has come a flurry of papers demonstrating that the performance improvements of RDMA stun at all levels of abstraction. At their lowest levels, traditional replicated systems rely on consensus protocols, most popularly the many variants of Paxos, to provide a totally-ordered stream of events. By adapting Disk-Paxos to RDMA hardware, a team led by Naama Ben-David demonstrated that even these decades-old protocols can enjoy order-of-magnitude performance improvements over RDMA [Aguilera et al., 2019]. Moving RDMA up the stack only brings more improvements; systems like Corfu and FARM have demonstrated that rethinking replicated storage itself in the presence of RDMA can yield

systems which process events at previously unheard-of rates [Balakrishnan et al., 2012; Dragojević et al., 2014].

We can also question another core assumption of consistent replication: that consensus is required at every step in the first place. Existing work makes it clear that significant exceptions to this assumption exist. For example, Google's Spanner leverages GPS and atomic clocks to keep replicas precisely synchronized, taking the "wall clock" time required by linearizability quite literally [Corbett et al., 2013]. But perhaps Spanner's more interesting innovation comes in their read-only transactions. In Spanner, read-only transactions *are allowed* to return slightly stale results—a seeming violation of linearizability. In justifying this choice, the designers of Spanner argue that, given the latencies involved in communication with clients in a distributed system, having a slightly stale read result returned to the client is no different than having an up-to-date result returned to a client living a few milliseconds further away. This argument is not dissimilar to our earlier reasoning about the key differences between consistency and integrity as properties of data: the moment data leaves a transaction its consistency begins to decay. By focusing their energy on transactions which write, Spanner's basic architecture—variations of which have since found their way into many other systems—is able to scale impressively for certain workloads.

Spanner effectively argues that the presence of some less-than-consistent system behaviors should be permitted if the result is indistinguishable (or nearly so) from fully consistent behaviors; they have (perhaps unwittingly) once again taken the question of consistency out of the system domain and into the domain of programming languages. The authors of Spanner are making an argument that properties of the programs written to run

against Spanner—in particular the latency between those programs and the Spanner system itself—composed with the nearly-linearizable guarantees provided by Spanner, results in an *externally* consistent program: one in which the visible effects of the program running against near-linearizable storage are in line with those expected under a truly-linearizable execution.

The key observation is that by limiting the language (e.g. by introducing latency in Spanner), one can reduce the guarantees provided by storage *without* changing the behavior of programs running against it. We can view the techniques of Bloom^L as an extreme example of this.

0.4.2 *Convergent Programming over RDMA*

These two different approaches—limiting the language vs reducing round-trip times—have demonstrated two very different ways to achieve consistency without the limitations imposed by traditional replication protocols. But even more performance can be unlocked by combining them. Chapter 2 of this dissertation explores leveraging monotonicity over RDMA in the construction of a new system: Derecho. Derecho's core component is a simple shared state table (SST) data structure for storing state replicated via RDMA. This structure is a table, but with a twist: each row of the table may be written to only by a dedicated node corresponding to that row. State is replicated by having each node directly write its assigned row to the table at all replicas, ensuring both that no write-write conflicts may occur, and that the cost of replication is borne only by the CPU of the writer; RDMA allows the write to skip the receiver's CPU entirely, depositing the write directly into memory. Derecho places one additional

limitation on the SST: the data it stores must be ordered, and writes to the SST must be *monotonic*. Any write performed to a cell of the SST must be of some value equal to or greater than the one it is replacing, according to some globally agreed-upon ordering.

Atop this SST sits a new core language for monotonic programming. The core of this language is a set of combinators used to define monotonic predicates over ordered data. These combinators are used to read values from the SST, transforming ordered datatypes—such as counters—into boolean observations. Because the combinators of this language are monotonic, and because the state of the SST itself evolves monotonically, all predicates written in this language are *stable*: once they become true, they remain true for the remainder of the program.

Using these monotonic predicates, programmers can also supply *triggers*, snippets of code to be executed when certain predicates become true. These triggers, too, are limited; they can read values from the SST, use monotonic combinators to transform them, and then write those values back elsewhere in the SST. Additionally, triggers may use a general-purpose language to perform arbitrary computations and produce externally-visible effects, but must ensure the computations they perform do not depend on weakly consistent values in the SST.

Much like its cousin Bloom^L, this highly limited language comes with a compelling correctness result: the same set of triggers will fire under a weakly consistent execution as would fire under a linearizable execution. This core language combines the speed of *both* RDMA-based replication *and* convergent asynchronous programming, providing an unrivaled substrate atop which to build programs.

But we cannot stop here. This language is too limited to be used to build realistic distributed applications; deploying it directly into the world would do nothing to stop the spread of more expressive, weakly consistent systems. There is however one "killer app" for which this language is quite well-suited: building the strongly consistent replication protocols themselves.

0.4.3 *Derecho: Datacenter Replication at the Speed of Light*

It turns out that the core problem of replication itself can be phrased as a sequence of stable predicates over monotonic, protocol-defined state. Leveraging this, one can build consistent replication atop this core SST language, ensuring that replicas process operations with minimal delay.

We do this in Derecho, adopting Ken Birman's virtual synchrony protocol [K. Birman and T. Joseph, 1987] to the SST and producing best-in-class data rates in the process. Beyond its novel synthesis of RDMA and monotonicity, Derecho also leverages the idea of control plane and data plane separation, implementing its data streaming protocol out-of-band with its control messages. Data managed by the system is replicated via RDMC, which implements pipelined reliable multicast atop RDMA [Behrens et al., 2018]. RDMC-delivered messages are then held at the receiver until the SST's consensus protocol confirms all correct replicas have received them.

Derecho's separation of data and control makes it especially well-suited to the domain of streaming computation or batched data processing. These systems are in increasingly high demand, as new sources of data—such as the cameras, drones, and monitoring systems that are quickly becoming

ubiquitous in all corners of life—overwhelm existing consistent systems, forcing users to fall back to weakly consistent systems to process this data. Current industry-standard strongly consistent systems have been overwhelmed by the data tsunami. Derecho is a system built to survive it.

Rather than provide a simple key–value store or the monolithic abstractions of a database, Derecho instead serves as a library via which replicated programs can be built directly. Derecho's core user-facing abstraction is of Replicated Actors [Hewitt, Bishop, and Steiger, 1973], managed by Derecho and used by programmers via a custom, RDMA-aware RMI framework. Derecho mixes this idea with the once-popular model of process groups, automatically allocating, partitioning, and decommissioning instances of these actors based on programmer-supplied constraints. These abstractions make Derecho almost a language itself: a DSL for replicated actors, embedded within C++.

Using Derecho's replicated actors, the Derecho team has built several example distributed systems, including the ever-popular key-value store and a distributed filesystem.

0.4.4 *The Limits of RDMA*

Within a single datacenter, the performance enabled by Derecho obviates the need for weak consistency, demonstrating that strongly consistent replication is alive and well. But modern distributed applications cannot always be confined to a single datacenter; and in the wide area, transport technologies have simply not seen the same pace of innovation. While RDMA over the wide area does exist, it is as yet not reliable enough for

use with Derecho; and while Derecho does have a TCP backend, its core components were carefully engineered for the setting of RDMA within a single datacenter. At present, Derecho over the wide area remains a work-in-progress; in this domain, weak consistency still rules.

At a large enough scale, the abstraction of lockstep replication will simply never *be* viable. Two replicas attempting to synchronize across continents will always need to contend with the speed of light, introducing a hard lower bound on the round-trip time required for consistent replication. In these settings, the latency of replication really *is* fundamental.

So we must again turn to that hallmark of weak consistency: asynchronous replication. But this time we must find a way to ensure the data we share stays consistent, despite asynchronous replication.

0.5 CONSISTENT PROGRAMMING OVER THE WIDE AREA WITH GALLIFREY

0.5.1 *Systems Solutions with CRDTs*

Existing attempts to build strong consistency on asynchronous replication have had mixed results. In the systems community, efforts to build correct programs with asynchronous replication have rallied around the idea of Commutative (then convergent, then consistent) Replicated Data Types (CRDTs) [Shapiro, 2017]. As most famously posed by Marc Shapiro, CRDTs are collections of data structures all of whose operations commute. This means that individual replicas will always converge to the same state, no matter the order in which they believe their operations occurred. This idea

does an end-run around the question of consistency itself; consistency focuses on the order operations are allowed to appear in, and CRDTs boldly declare that this order doesn't really matter. As CRDTs have gained popularity, they have been introduced as core primitives in many weakly consistent systems like Cassandra and MongoDB, [Lakshman and Malik, 2010; Plugge, Membrey, and Hawkins, 2010]. These systems tend to offer CRDTs alongside traditional weakly consistent registers, but increasingly, whole systems are being built around CRDTs and CRDTs alone [Akkoorath et al., 2016].

The promise of integrating CRDTs into weakly consistent storage systems is clear. The worst danger of weak consistency is the lost write; the possibility that a critical operation will simply disappear from a system, having been "overwritten" by some event from arbitrarily far into the future or long into the past. With CRDTs that fear vanishes; the guarantee of commutativity *also* provides a guarantee that overwriting simply doesn't happen. Think about it: if some operation o_2 overwrites some operation o_1, then that implies that o_1 happened before o_2. This in turn means that o_1 and o_2 can't commute; if we were to reverse their apparent order, then the apparent overwriting should go away.

While not all data structures feature fully commutative operations, one in particular does: the grow-only set. Catchy example CRDTs, like the shopping cart from DynamoDB [DeCandia et al., 2007] or vote tallies from my own work [Milano and A. C. Myers, 2018], all tend to boil down to sets which only grow: for example with maps implemented as sets of pairs, counters implemented as sets of increment operations, and shopping carts implemented as sets of items. Even CRDTs which permit removal *still* tend to be represented by sets that only grow; rather than have a single set, you

have two: one for items to add, and one for items to remove. It's up to the reader of those sets to decide what to do with them.

The problem with CRDTs, as it turns out, is reading them. CRDTs guarantee that all operations commute; they do *not* guarantee that any particular set of operations will be visible at any particular replica within any particular time frame. This means that the reader of a CRDT is still left with weak consistency. And if that reader uses its weakly consistent observations to then issue CRDT operations, the illusion of the consistent CRDT is shattered; suddenly there can again be "writes" which are not justified by any sequential order of operations in the system. This is where the harsh limitations of Bloom^L and Derecho's core language come from. It's not enough for the program to issue commutative operations; it must also ensure that the observations which *caused* those operations themselves commute.

Without resorting to language limitations, the maximum consistency that a system under fully asynchronous replication can offer is a variation on causal consistency [Mahajan, Alvisi, Dahlin, et al., 2011]. This consistency corresponds to a full information protocol; it's what you get if every node is constantly gossiping its entire message history and local state to every other replica it can find[3]. The native CRDT of causal+ consistency is the partially-ordered log; systems like fuzzylog or Corfu offer this log abstraction directly—with the blazing speed of weak consistency of course—and it too has proven a popular abstraction against which to build distributed programs [Balakrishnan et al., 2012; Lockerman et al., 2018]. If you're willing to make some semi-synchronous or some high availability

3 Of course, real implementations are substantially more efficient than this naive protocol implies

assumptions, you can improve on causal consistency a bit with prefix consistency, which promises that all unknown operations occur within a single "gap" in the operation history visible to a client; it's usually phrased as a log in which all clients know some prefix (the global log) and some suffix (their pending operations), with a gap in the middle [D. Terry, 2013].

Combining CRDTs with causal+ or prefix consistency unquestionably improves the reliability of distributed programs. But it's *still not strong consistency*. Bizarre errors can (and do) still occur, and the programmer still has no access to that basic reassuring metaphor of monolithic memory, sequentially processing events.

So we must turn once again to the language side of this problem, and the idea that limitations to the expressiveness of the language can allow for consistent programming over asynchronous replication. We can lift the simple languages that thrive in this space—like Bloom^L or the SST's monotonic core language—into a general-purpose setting, and make them viable for large-scale, wide-area distributed application development.

0.5.2 *Gallifrey: A New Language for Geodistributed Programming*

The key to building a consistent language over asynchronous replication is to drill down on the core weakness of current CRDT use: using the exact (weakly consistent) value of a CRDT in determining which CRDT operations to issue. This sounds like something we've heard before, twice now. MixT argues that allowing weakly consistent reads to influence strongly consistent writes can lead to errors; now we see the error is allowing weakly consistent reads to influence *any* writes. But some weakly consis-

tent reads are safe: in Derecho's core, the SST's language of combinators expresses *stable predicates* over *monotonic data*, ensuring actions triggered by these predicates are safe to perform even if the underlying monotonic data is not strongly consistent. Under Derecho's model, we learned that general-purpose computation which depends only on the existence of some true predicates will always be safe to perform.

This, then, is our plan. We integrate a language of monotonic mutations and transformations—like the core language of Derecho—into a general-purpose, imperative programming language, allowing both monotonic mutations to shared data and predicate/trigger-style observations of it. And like MixT, we employ a type system to ensure that these limits are respected: that the general-purpose computations we perform do not normally observe our weakly consistent state, and that the code which has access to weak consistency does not influence the remainder of the program—except through stable predicates over monotonic state.

But it's not enough to stop there. Simply promising a type system for the safe use of CRDTs will not satisfy the needs of distributed programmers. CRDTs alone have not proven expressive enough in the past; limiting them is unlikely to improve matters. We must combine our safe CRDTs with a more general-purpose—and necessarily synchronous—mechanism for sharing arbitrary data, allowing for patterns—like locks—which are simply not possible under weak consistency.

The solution is to erase the distinction between objects managed by the system and those native to the program. To eliminate the idea that there are a finite set of CRDT types, and replace it with a unified model of objects, some of which may be replicated, working together in a traditional, Java-like object-oriented programming language. What we propose is a notion

of *orthogonal replication*: that at any point, any traditional sequentially-specified object may be shared, or any existing shared object may be reclaimed as a sequential one.

Enabling this idea are two key new language features: *Restrictions* (in chapter 3, with Gallifrey) and an *isolation* type system (in chapter 4). Restrictions refine the interface of a sequential object, identifying a subset of the object's methods which are safe to call concurrently. Restrictions effectively identify the CRDT hiding within a sequential object, and expose only the operations of this hidden CRDT. If the operations exposed by a restriction correspond to a set of monotonic mutations, then restrictions also allow programmers to specify *tests* over their shared object. Much like the predicates within Derecho, these tests are monotonic computations over shared state, and may be used to define triggers which fire once the predicate becomes true. Taking a queue from Lasp [Meiklejohn and Van Roy, 2015], Gallifrey's predicates and triggers have the form of a `when(predicate) trigger` statement which blocks until the `predicate` is true.

Restrictions are not limited to just sharing traditional CRDTs, however. Restrictions also allow programmers to expose even non-commutative (or non-monotonic) operations on shared data, requiring that programmers supply a `merge` function to resolve any conflicts which may occur during runtime. By ensuring these operations cannot influence any exposed observations, Restrictions know that allowing such operations will not weaken the consistency of the overall application. When an object outgrows the bounds of a single restriction, programs may *transition* it into a new restriction. This requires consensus; before these operations are allowed to occur, the system as a whole must agree on the "true"

value of the underlying object. Using these mechanisms, programmers can implement synchronous behavior—like reading exact values—atop asynchronously-replicated objects.

0.6 REASONING ABOUT ISOLATION WITH STATIC TYPES

Embracing the idea of sharing *any* object raises an important question: where does the sharing stop? It is not sufficient to just share the literal structure in question without also sharing its reachable object graph: the internal references and objects that are used by the shared object's methods, and that together form the actual abstraction of the shared data structure. But in a traditional object-oriented language the notion of "internal object graph" is ill-defined; in principle any object, anywhere, can refer to any other object without limit. This is unworkable for restrictions; in order for programs which use restrictions to be correct, Every access to a shared object *graph* must occur via a restriction. We cannot enforce this unless we know that the objects guarded by a restriction are not otherwise reachable; or at least we must know how they may be otherwise reached, and prevent that access.

0.6.1 *Existing Type Systems*

Variations on this problem are quite well-studied in the programming languages literature. We could cast this problem as a memory safety issue: we could treat sharing an object under a restriction as manually deleting it and then allocating a new `shared` object, and look to languages with

safe manual memory management to provide the guarantees that we want—that no references to the pre-shared state of the object may exist. Here we could turn to projects like Cyclone and Rust, which promise full memory safety and thus could be used as a proxy to achieve what we want [Fluet, Morrisett, and Ahmed, 2006; *Rust Programming Language* 2014]. Some of these languages—Rust chief among them—have actually recognized this connection and explicitly extended their memory safety guarantees to thread safety, a much closer match for the property we need. But these languages are often very low-level; they frequently either require excessive programmer annotation or are limited in their expressive power. These languages are easy to use when the object graph forms a forest, but range from difficult to impossible (without unsafe code) to use when one needs to represent a more general graph. Moreover, manual memory management has an escape hatch we can't rely on: the ability to simply *not* delete an object when references to it might still exist. In a memory management setting this causes leaks, but doesn't change the semantics of the language much. In our setting it would correspond to making certain objects unshareable—a major, and visible, limitation.

We could also cast this problem as one of ownership; in this setting, we could say that a shared object must "own" its reachable object graph, and use an ownership type system to prevent external accesses to these objects. Ownership type systems—popular in the early 2000s—tend to be implemented via extensions to Java, making it a much closer match to the level of abstraction we wish to offer in Gallifrey [Boyapati and Rinard, 2001; D. Clarke, Wrigstad, et al., 2008; D. G. Clarke, Potter, and Noble, 1998]. But these systems don't always introduce the possibility to *change* owners, a feature that Gallifrey—with its ability to share and transition any object—

will require. And systems which do allow ownership changes tend to rely on one feature of Java that we *don't* want to include—the potential for any reference to be null [Aldrich, Kostadinov, and Chambers, 2002; Boyapati and Rinard, 2001]. By identifying and nulling out dominating references, these systems can effectively sever part of the object graph, ensuring that any attempt to reference moved memory will hit an exception instead. But this forces programmers into a pattern of treating every reference as potentially null, clogging up code with dynamic checks or subjecting it to unexpected runtime failures.

Many other approaches exist: capability systems which separate the idea of access to a reference from simply holding it [Clebsch et al., 2015; Haller and Odersky, 2010], region-based systems that generalize single objects into object subgraphs and treat them as a unit [Fluet, Morrisett, and Ahmed, 2006; Tofte and Talpin, 1994], or pure linear systems that treat references as resources [Fähndrich and DeLine, 2002; Wadler, 1990] to name just a few. But none feature the right mix of ease-of-use and expressive power to make them a good match for static restrictions on shared objects.

0.6.2 *Isolation Types*

At their core, each of these systems proposes a different way of approximating precise reference tracking. Ideal reference tracking—the ability to statically know exactly which objects are referred to by exactly which references—would provide perfect information for memory management, concurrency, or abstraction defense. In our setting, it would mean knowing

precisely which objects are connected to a given restriction, and knowing precisely which references outside of this restriction refer to objects within it. With this information in hand we could simply make it a static error to use those references, and have a perfect defense of Restrictions as a result.

The trouble is that inferring these relationships is undecidable in general. To make it tractable, type systems need to employ a mix of user-provided annotation and limitations to the form of the object graph; a huge design space which, while heavily explored in the past, still has much ground yet to cover. Existing solutions tend to lean heavily in one direction or another, allowing arbitrary object access while requiring the user to reflect possible access patterns into the types (as in much of the region work), or sticking to relatively harsh limitations on the shape of structures permitted so as to avoid pervasive annotations (as in Rust). No practical language is completely at either extreme, and some efforts have been made to find compelling middle ground.

Chapter 4 of this thesis represents a new effort to find a middle ground. Like much of the work that has come before, chapter 4 presents a type system which manages memory at the level of *regions* [Tofte and Talpin, 1994]. The set of objects in a region is statically known, but the object *graph* within that region—the set of edges connecting those objects—is opaque to the type system. Instead, the type system tracks only the references which cross regions, ensuring that the *region graph*—where an edge exists between two regions if an object in one region is connected to an object in the other under the object graph—is statically known. When an object is shared under a restriction, the type system must ensure that the object graph reachable from the newly-shared object is *only* reachable via that object. The type system approximates a shared object's reachable object

graph with its reachable region graph; it can then conservatively assume all objects in those regions are now governed by the restriction, and prevent any external access to them. This ensures restrictions are respected.

But in doing so, it raises the potential for significant user annotation: we must know which objects live within each region, and we must know how those regions are connected. While this can often be inferred within functions, the problem of inference runs into a wall when paired with interfaces in a traditional, Java-like programming language. It's just not possible to infer the region expectations of a method from its type signature alone—which is all that is available when programming against interfaces.

All hope is not lost, however. In our domain, we primarily wish to identify the graph of objects programmers intend to share. By their nature these object graphs will likely *already* be nearly-isolated; years of experience managing objects as messages and working with traditional object graph serialization has taught programmers that abstractions intended for concurrency—whether as actors, monitors, or messages—ought to have few external references, and ought to precisely control those references which do exist. Recognizing this existing pattern, we define a notion of a *simple* region as one whose reachable region graph forms a tree, and a *simple* object as one whose directly accessible regions are all simple. Programmer annotations are required only to capture deviations from simplicity, which we anticipate will be relatively rare.

By combining this new type system with the restrictions presented in the previous section, we can achieve our goal: writing consistent-by-construction programs against weakly consistent replication.

0.7 COMMON THEMES

In the previous sections, we introduced three distinct ideas to conquer the challenge of building correct distributed programs at the scale the world requires. In MixT, we lent sanity to the current zoo of multi-store, multi-consistency systems, reducing our problem to the more tractable (but still difficult!) task of programming against single-consistency systems. With Derecho, we demonstrated that leveraging emerging developments in both network technology and programming languages yields a strongly consistent system capable of unrivaled data processing speeds within a single datacenter. In Gallifrey (and its associated isolation type system), we escaped the bounds of a single datacenter and built a language for writing consistent distributed programs which span the globe.

Uniting these approaches is a common thread: viewing replicated data not as a single abstraction defined by its consistency, but as one part of a program which can be correct—or incorrect—independently of the consistency of its underlying replication. Rather than change the guarantees provided by the system, the key is to change the guarantees provided by the language we use to program it.

We see this in MixT, which assumes that programmers can use consistency guarantees as a proxy for their data invariants, and provides a typed language for transactions which ensures that these invariants are respected. Without MixT, the presence of a single weakly consistent object in a program is enough to threaten the consistency of the entire program; with MixT, that potential inconsistency is contained.

We see this even in Derecho, a system for programming against strongly consistent replicated actors. Derecho's core protocol—the source of both its speed and its consistency—is itself implemented atop weakly consistent replicated state, built in a core language which leverages monotonicity to eliminate an entire category of errors caused by weak consistency.

And we especially see this in Gallifrey and the type system which enables it. Gallifrey does not reason about the consistency of replicated objects at all; through its Restrictions, Gallifrey hybridizes the monotonic properties of Derecho's core language with the replicated actor model of Derecho's surface, yielding a general-purpose language in which consistency is guaranteed over arbitrarily inconsistent replication.

Correct programs can be written against weakly consistent state. Doing so today relies on complex engineering, careful invariants, and a long list of "best practices." MixT, and Gallifrey aim to replace these ad-hoc invariants and "best practices" with sound, language-based guarantees, giving programmers clear answers to when their programs are correct despite the underlying core of inconsistent replication. Derecho leverages those same guarantees to rebuild the very abstractions we are replacing, demonstrating that using a language-based approach to consistency does not just yield more correct programs, but more performant ones as well.

0.8 A ROADMAP

The rest of this dissertation proceeds as follows. Each of the next four chapters will be devoted to a deep dive into one of this dissertation's constituent systems: first MixT, then Derecho, then Gallifrey, and finally

the reservation-safe type system that makes Gallifrey tick. These chapters are self-contained, including the relevant background and related work necessary to understand their contributions in isolation. Finally we conclude by viewing these systems together, and see how each may continue to inform the development of these ideas into the future.

1

MIXT

1.1 INTRODUCTION

This chapter introduces *mixed-consistency transactions* with MixT. Mixed-consistency transactions allow programmers to combine the blazing speed of weak consistency with the safety and clear semantics of strong consistency. This improves on existing approaches, which support transactions that operate with a single consistency model at a time.

Traditional strongly-consistent tools, such as strictly serializable atomic database transactions and distributed locking, do not scale across continents; the speed of light simply isn't fast enough for the cross-continental round trips needed by traditional transactions. Newer weakly-consistent tools enable lower latencies and higher availability at the price of weaker guarantees. Evidence from both industry [Carlson, 2013; Gentz et al., 2017; Klophaus, 2010; Lakshman and Malik, 2010; Plugge, Membrey, and Hawkins, 2010] and academia [Crooks et al., 2016; Holt, Zhang, et al., 2015; Sivaramakrishnan, Kaki, and Jagannathan, 2015] suggests that weak consistency can be viable for some data, while other data needs stronger consistency—implying that single applications can need multiple *levels* of consistency.

One way to work around this is to run applications at a single consistency level strong enough to enforce all required data invariants. But

running with a single consistency level, as seen in prior systems [Brutschy et al., 2017; Dongol, Jagadeesan, and Riely, 2018; Kaki et al., 2017; Xie, Su, Kapritsos, et al., 2014; Yang, You, and Gu, 2017; Yu and Vahdat, 2000]), can be slow [Brewer, 2010]; all operations within a transaction must be upgraded to the consistency required by the most sensitive among them, introducing unnecessary delay and contention for objects that only require weak consistency. This is a fundamental problem: we need a general way to construct transactions that access data at multiple consistency levels, without compromising strong consistency where it is needed.

These concerns lead us to a key observation: consistency is a property of information itself and not only of operations that use this information. Further, the consistency with which we manipulate information should always match or exceed the consistency at which we store it. This observation forms the basis of *mixed-consistency transactions*. Each mixed-consistency transaction can operate over any and all data, even if this data is stored with varying consistency guarantees. Despite this expressivity, we maintain the consistency guarantees required by each data object by preventing less-consistent information from influencing more-consistent data. Mixed-consistency transactions are atomic: no operations inside a transaction become visible outside the transaction until all operations do.

In implementing mixed-consistency transactions, we uncover a further complication: engineers at major companies frequently find themselves writing distributed programs that mix accesses to data from multiple existing storage systems, each with distinct consistency guarantees [Brown, 2017]. It is unrealistic to assume that data can be freely migrated into ever-newer and more capable storage systems, or that all applications can be written against a single unified system; we therefore want to

operate against multiple backing stores within the same mixed-consistency transaction.

MixT addresses these challenges in a single solution: a domain-specific language for mixed-consistency transactions. In MixT, persistent data and operations at various stores can be accessed with strong guarantees (§1.3). To ensure the semantic guarantees of mixed-consistency transactions, weaker-consistency information should avoid influencing stronger-consistency information. To prevent this influence, MixT views consistency as a property of data, treating consistency as a form of data integrity [Biba, 1977] expressed as labels on types in the language. Static analysis of information flow [Sabelfeld and A. C. Myers, 2003] then ensures that consistency guarantees cannot be violated by exposure to objects with weaker consistency.

The MixT language implements mixed-consistency transactions using three novel mechanisms (§1.4–1.5):

- Compile-time information flow control ensures that the consistency of data is never weaker than the level described by its storage location.
- Using information flow analysis, the code of each transaction is automatically *split* into sub-transactions for each consistency level, while preserving atomicity.
- A lightweight run-time mechanism ensures transactional atomicity, even between sub-transactions executing on multiple mutually unaware backing stores.

MixT works on top of stores' existing transactional mechanisms, without changing the representation of existing data, allowing existing applications

to operate unmodified alongside MixT applications. And MixT can be easily adapted to a new store, by inserting the store's consistency level into MixT's consistency lattice and providing bindings to custom data operations specific to that store.

As we show experimentally (§1.9), mixed-consistency transactions perform well. MixT enables significant speedup vs. serializable transactions by exploiting weak consistency, without losing the guarantees sacrificed by current systems when consistency levels mix.

1.2 MOTIVATION

1.2.1 *A Running Example*

Suppose we are building a scalable group messaging service called *Message Groups*. This service allows users to join groups and to post messages to all members of any group they have joined. For low-latency communication, application servers are deployed across the world, with data replicated across these servers.

Communication latency between these servers makes it difficult to keep the replicated data fully consistent without degrading user experience. Fortunately, there is no need to enforce a global, total order on displayed messages. It is only necessary to respect potential causality, so that messages precede their responses. We therefore geo-replicate user inboxes at a weaker consistency level, *causal+ consistency*, which respects causality but does not guarantee a total order [Lloyd et al., 2011].

However, other data in this application requires stronger consistency. The membership of users in various groups should be consistent worldwide so that all servers agree on who is supposed to receive which messages. Therefore, the set of members of each group is placed at a store supporting *linearizable transactions*, which ensure serializability and external consistency [Corbett et al., 2013; Herlihy and Wing, 1988; Papadimitriou, 1979]. Latency to this single store is necessarily much higher for many users than latency to their own inboxes.

1.2.2 *The Need For Mixed-Consistency Transactions*

To see the pitfalls inherent to this naive mixing of consistency levels, consider how Message Groups might implement logged message delivery, using the code in fig. 1.1. There is a linearizable list of members named `users` to whom a message `post` should be sent. Each member in `users` has a causally consistent inbox. For regulatory compliance, the sending of messages is logged (via `log.append`). The log does not even require causal consistency; instead, we might replicate it at *eventual consistency* [D. B. Terry, Theimer, et al., 1995], which requires only that reads converge after a sufficiently long quiescence. All mutations, including append and insert, are replicated across continents.

However, the simple loop in fig. 1.1 does not address concurrent modification to the data. Suppose that a thread concurrently modifies `users` while another thread is executing fig. 1.1. Without care, this concurrent modification might invalidate fig. 1.1's iterator; at best, it is unclear whether

```
var iterator = users,
while (iterator.isValid()) {
  log.append(iterator->v.inbox.insert(post)),
  iterator = iterator->next
}
```

Figure 1.1: Naive code for sending messages. Corrected MixT code is found in fig. 1.4.

the new member will receive a message. As written, there is no reason to expect the result of this execution to be atomic, isolated, or even complete.

Clearly, some form of concurrency control is needed. We might change over to a recent system such as Quelea [Sivaramakrishnan, Kaki, and Jagannathan, 2015] or Salt [Xie, Su, Kapritsos, et al., 2014], which provide both fully atomic transactions and multiple consistency levels. But these systems can only execute a given transaction at a single consistency level. In these systems, all data in the Message Groups example would effectively be upgraded to linearizability. There would be no performance benefit from having a weakly consistent inbox and log; message delivery performance would likely be unacceptable.

Alternatively, we could partition our data onto three distinct systems, each optimized for the appropriate consistency level. If we had a causal store such as TaRDiS [Crooks et al., 2016] that supports interactive atomic transactions, we could start a separate simultaneous transaction at each system. This would achieve the desired performance, but the implicit interactions between these transactions could create bugs. For example, if another process updated the membership list while mail was being sent, the linearizable transaction might abort and roll back, restarting the loop. Without code to explicitly roll back the other concurrent transactions—which may not even always be possible—some users could then receive

the same message a second time. We might think to fix this problem by adding a "delivered" flag to each message, to be set when the message is sent. If the flag is linearizable, transaction rollback can reset its value and messages will still be sent twice. If the flag is causal, the programmer has to be careful to update it only at the end of the causal transaction, because causal updates might not be rolled back if the transaction retries, leading to lost messages. And even if we were to carefully solve this interaction, there is still the matter of the logger, a component that requests even weaker consistency (and expects correspondingly faster access).

Thus, this transaction cannot be naturally implemented using each underlying store's mechanisms in isolation. It requires a new form of *mixed-consistency*, *mixed-location* transaction not supported by any existing system, with new run-time mechanisms for atomicity across different consistency levels.

1.2.3 *Mixing Consistency Breaks Strong Consistency*

There is a reason that existing systems choose a single consistency level for each transaction. Causal consistency and linearizability offer well-defined consistency guarantees, but trying to mix these levels in the same application can break the guarantees that both levels claim to offer, even if all the issues in the previous section were solved. Some transactions simply are not safe to run under mixed consistency. To see why, consider the following example.

Suppose we run a contest to advertise Message Groups. Users are divided into two teams; team A sends messages to mailbox `a`, and team B

```
if (a.inbox.size() >= 1000000 &&
   b.inbox.size() < 1000000) { //weak condition
  a.declare_winner()       //strong effect
} else
if (a.inbox.size() < 1000000 &&
   b.inbox.size() >= 1000000) {
  b.declare_winner()
}
```

Figure 1.2: Contest logic inside the mail delivery transaction. Corrected MixT code is found in fig. 1.15.

sends messages to mailbox `b`. The first team to send 1,000,000 messages is declared the winner.

To implement this contest, we extend the existing transaction for delivering mail with a few lines of code shown in fig. 1.2. After running the contest, we may be surprised to discover that the code has not declared a unique winner; both teams A and B are simultaneously declared the winner!

This code has a fundamental problem. To avoid slowing down the core functionality of message delivery, the guard condition uses data (the inbox sizes) stored with only causal consistency. Since the guard is evaluated with causal consistency, nothing guarantees that the function `declare_winner()` is invoked only once. But the function `declare_winner()` manipulates only data with linearizable consistency; it is not designed to deal with the potential for multiple re-executions. During a partial network partition, every single message receipt to either team could cause the winner to switch, as the causal replica receiving messages for `a` may not be able to propagate events to the replica receiving messages for `b` (and vice-versa). This causes each team to believe their inbox alone has reached the target size.

The essence of this mistake is that more-consistent data (the declared winner) is influenced by less-consistent data (the inbox size). This inappropriate influence means the developers of `declare_winner()` would have to add complex code to ensure its assumptions hold under weak consistency guarantees, even though `declare_winner()` does not access weakly consistent data itself.

The issue of weakly consistent data influencing strongly consistent computations is fundamental to the semantics of consistency. Even within a linearizable transaction, the influence of weakly consistent data on program control flow can effectively weaken the isolation level of the entire transaction. MixT uses information flow analysis to flag such influences at compile time, disallowing this example code. As discussed in section 1.7, MixT also allows intentional weakening of this restriction.

1.3 MIXT TRANSACTION LANGUAGE

We solve the problems introduced in the previous section with MixT, a new domain-specific language (DSL). To support a variety of underlying stores in a uniform way, including key–value stores, databases, and file systems, MixT offers a high-level embedded transaction language that is straightforward to adapt to new stores.

$$
\begin{aligned}
x &\in \textbf{Var} \qquad f \in \textbf{Operation} \\
\oplus &\in \textbf{Binop} \qquad \ominus \in \textbf{Unop} \\
(\textit{Location})\ m &::= x \mid *e \mid e.x \mid e\texttt{->}x \\
(\textit{Expr})\ e &::= m \mid e_1 \oplus e_2 \mid \ominus e \mid e_0.f(e_1, \ldots, e_n) \\
(\textit{Stmt})\ s &::= \texttt{var}\ x = e \mid m = e \mid \texttt{return}\ e \\
&\quad \mid \texttt{while (}e\texttt{)}\ s \mid \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \mid \texttt{\{}s_1, \ldots, s_n\texttt{\}}
\end{aligned}
$$

Figure 1.3: MixT surface syntax. Certain built-in operations are omitted for clarity of exposition.

1.3.1 *MixT Language Syntax*

fig. 1.3 gives the surface syntax of the MixT language. Because MixT is embedded in C++, its syntax and semantics, though different from those of its host language, are designed to be unsurprising to C++ programmers.

MixT is relatively expressive; for example, it has control structures like conditionals and loops. Despite supporting real control structures, MixT transactions are fully atomic when the underlying stores support atomic transactions. In particular, all transaction effects become visible at once, and transactions operate against stable snapshots at each store. Like C++, the MixT language has mutable locations, which can be either local variables or fields of objects.

Though they are not shown explicitly in fig. 1.3, *handles* are a key abstraction of MixT. Handles behave like pointers to remotely stored persistent data; they can be dereferenced to access the underlying data (with the operators $*$ and `->`), and they can be aliased by assignment.

Handles also support the invocation of operations on data. Given a handle e_0 to a receiver object, the expression $e_0.f(e_1, \ldots, e_n)$ invokes a

```
class user {
 Handle<set<string>, causal, supports<insert>> inbox;
};

class group {
 RemoteList<Handle<user, causal>, linearizable> users;
 Handle<Log, eventual, supports<append>> log;

 mixt_method(add_post) (post) mixt_captures(users,log) (
  var iterator = users,
  while (iterator.isValid()) {
   log.append(iterator->v.inbox.insert(post)),
   iterator = iterator->next
  }
 )
```

Figure 1.4: MixT message delivery implementation (§4.2). MixT code (lines 8–14) is colored green; C++ code is blue.

custom operation named *f*, provided by the underlying store of the receiver.[1] Exactly which operations are supported depends on the store. For example, many stores provide specialized operations for manipulating sets, but even SQL queries can be exposed as operations.

1.3.2 *A MixT Example Program*

As an example of MixT code used within a larger program, the message delivery transaction of section 1.2.2 is shown in fig. 1.4. To distinguish MixT code from surrounding C++ code, MixT code is colored green, whereas C++ code is blue.

1 The store may specify whether its parameters should be treated as opaque handles, arbitrary values, or dereferenced handles to other objects on this store. The arguments $e_1, \ldots, e_n$ are passed as values by default, except when the store requests otherwise, at which point they will be dereferenced (resulting in a run-time error if these arguments refer to handles on the wrong store).

Most of this code should look familiar to a C++ programmer; outside the transaction, it merely defines classes that contain library types, such as the MixT library type `RemoteList`, as fields.

At the heart of MixT are transaction blocks, signified by the `mixt_method` declaration. For example, at lines 8–14 is the now-familiar transaction for message delivery, expressed as a method `add_post()` of the C++ class `group`. This method can be invoked from any context without the need to explicitly start a transaction; its parameter `post` is automatically inferred to be a string.

In this transaction, the expressions `iterator`, `inbox`, `users`, and `log` are all handles for state on remote stores. The type `Handle<T, L, ...>` is the C++ representation of a MixT handle for data of type `T`, stored at consistency `L`. An object of this class acts as an opaque representation of a persistent resource. Any supported custom operations appear in the third and following argument positions. For example, `inbox` (line 2) is a set of strings, stored at causal consistency, with a custom operation `insert` for adding new items to the set. It is the job of the causal store to ensure that `insert` operations from different clients are merged with causal consistency.

MixT offers some useful data structures as library types. For example, the type `RemoteList`, used at line 6, is a persistent linked list that stores its spine at a specified consistency level (here, linearizable).

```
class Handle<Type, Label, Operations...> {
   Type get(TransactionContext);
   void put(TransactionContext, Type);
   bool isValid(TransactionContext);
   Type clone(TransactionContext);
};

class DataStore<Label> {
   TransactionContext beginTransaction();
};
class TransactionContext {
   bool commit();
   DataStore store();
};
```

Figure 1.5: Handle and DataStore interfaces.

1.3.3 *MixT API*

As much as possible, MixT operates as a shim above existing stores, reusing their existing mechanisms for replication and data consistency. It is straightforward to add support for a new store, as long as it offers the necessary functionality; one simply implements three interfaces, `Handle`, `DataStore`, and `TransactionContext`, shown in fig. 1.5.

The `Handle` interface consists of a simple get/put/check API for accessing underlying data, a set of routines for supporting marshaling, and a set of routines for accessing and using the store from which the `Handle` originated. Much of this functionality can be automatically generated by the MixT libraries at compile time; fig. 1.5 only includes routines the programmer must implement.

A `DataStore` serves as an entry point to the underlying storage system; it is always associated with a specific consistency label (level) and a specific implementation of `Handle`. The only requirement from the `DataStore` API is the method `beginTransaction()`, which must create a new transaction

```
class CausalStore : public DataStore<causal> {
  template<typename T>
  class CausalObj : CausalHandle<T> {...};
  template<typename T>
  mixt_operation(insert) (CausalObj<set<T>>&, T&) {...}
  ...
};
```

Figure 1.6: Implementing a causal store in a host C++ program.

represented by a `TransactionContext` object. The `TransactionContext` can be used to commit or abort the transaction interactively, and can be extended to supply store-specific transaction interactions options. A `DataStore` may also implement any number of custom operations, ranging in complexity from creating new remote objects to processing SQL statements.

fig. 1.6 illustrates how a causal store with a custom operation `insert` can be implemented. Custom operations are declared within classes implementing `DataStore` by using the `mixt_operation` keyword. In this example, the operation `insert` is declared to take a remote `set` and a local `T`, matching the types on which it was invoked in `add_post` (fig. 1.4). Unlike `mixt_method`, `mixt_operation` does *not* declare a C++ method, and can only be called from within a MixT transaction. Within a transaction, operations dynamically dispatch to the appropriate `Handle` and `DataStore`; to facilitate this dispatch, every `Handle`'s type also includes a static list of operations which its implementation supports.

MixT custom operations provide a method-like syntax for invoking operations directly on handles to remote data, as with `insert` in fig. 1.4. It would be a mistake, however, to imagine that they are limited only to "method-like" invocations directly on remote data; they are flexible enough to expose arbitrary database functionality directly to a MixT

transaction. For example, one could create a `Handle<DB>`, with matching `mixt_operations` for interfacing directly with the underlying database's raw API. If the database exposed more stateful functionality, such as locks, a `Handle<DBLock>` could be used to manage each individual lock.

1.4 MIXED-CONSISTENCY TRANSACTIONS

section 1.2 shows that even seemingly trivial code can require the implementer to reason very carefully about the interactions between different consistency levels in the presence of possible transaction aborts. The complexity of this reasoning can easily become overwhelming. MixT tames this complexity by providing semantics for mixed-consistency transactions (§1.4.1). MixT's transaction support can provide atomic execution for section 1.2.1's message delivery transaction (§1.4.5), and its type system will detect the fundamental errors of section 1.2.3's contest (§1.4.3). A more detailed look at MixT's transactions comes in section 1.5.

1.4.1 *Defining Mixed Consistency*

We now address a fundamental question: what are the desired semantics of mixed consistency? We choose the standard approach used for shared-memory consistency [Herlihy and Wing, 1988], in which a consistency model is characterized as a trace property: that is, as the (possibly infinite) set of execution traces that do not violate the consistency model's guarantees. In principle, we can then verify whether a program execution satisfies a given model by checking whether its trace is in the set.

In mixed-consistency transactions, objects labeled with some consistency model should enjoy at least the guarantees of that model. For example, in a system with both eventual consistency and linearizability, traces involving any subset of objects should adhere at least to eventual consistency, and traces involving only its linearizable objects should respect linearizability. Put another way, an observer who accesses only linearizable objects should be unable to determine that there are any eventually consistent operations in the system.

The strength of consistency models can be characterized in terms of the possible behaviors of programs. The behaviors of the programs form a set of admitted traces T. The meaning of a consistency level ℓ is given by its consistency model, a set of traces $T_\ell \subseteq T$. A model T_ℓ is stronger than a model $T_{\ell'}$ when $T_\ell \subseteq T_{\ell'}$; T_ℓ provides more guarantees than $T_{\ell'}$. All consistency models must include the empty trace. We assume there is a lattice of consistency levels $\mathcal{L}$ ordered by strength. If a consistency level ℓ is stronger than or equal to another, ℓ', we write $\ell \sqsubseteq \ell'$. Consistency models are ordered by inclusion consistently with the ordering on $\mathcal{L}$: $\ell \sqsubseteq \ell' \iff T_\ell \subseteq T_{\ell'}$, $T_{\ell \sqcup \ell'} = T_\ell \cup T_{\ell'}$, and $T_{\ell \sqcap \ell'} = T_\ell \cap T_{\ell'}$.

Each trace $t \in T$ is a sequence of events e. An event e is a 5-tuple (a, o, ℓ, v, S) containing a, the action corresponding to this event; o, the exact memory location or object referenced by this event; ℓ, the consistency level of the store for this event's location; v, a tuple of any values processed by this event; and S, the client session in which this event occurred. Given such an event, we define $\mathit{consistency}((a, o, \ell, v, S)) = \ell$. For example, the program `x = 4; x = x + 1`, wherein `x` resides on a store with consistency ℓ, admits the trace "(write, x, ℓ,(4), S); (read, x, ℓ, (4), S); (write, x, ℓ, (5), S)" when executed in session S.

Given a trace t, the events relevant to a given consistency level ℓ are those whose consistency level is at least as strong. We write $t \downharpoonright \ell$ to denote the trace containing such events:

Definition 1.4.1 (Trace projection)**.**

$$t \downharpoonright \ell = [e \mid e \in t \wedge \mathit{consistency}(e) \sqsubseteq \ell]$$

Definition 1.4.2 (Mixed consistency)**.** A trace t exhibits *mixed consistency* if it satisfies *every* consistency model T_ℓ when projected onto that consistency level:

$$\forall \ell, t \downharpoonright \ell \in T_\ell$$

This definition is sensible even when working with incomparable consistency models; because consistency models form a lattice [Viotti and Vukolić, 2016], there is always some minimum consistency model onto which all events can be projected.

Definition 1.4.2 can also be adapted to transaction isolation levels [Berenson et al., 1995] by considering traces containing explicit events that begin and end transactions. A full formalization is found in section 1.6.

1.4.2 *Noninterference for Mixed Consistency*

In section 1.4.1, we proposed a definition for *mixed consistency* based on the approach used for shared-memory consistency. But this approach can hide influence: common consistency models, expressed in terms of

reads and writes to shared registers, are not strong enough to capture *why* each read or write occurs. To capture this influence directly, we look to *noninterference*, a semantic property common in the security and privacy literature. Noninterference describes programs as secure if, when given a policy lattice of security labels, program behavior at one point in the policy lattice cannot influence behavior at levels that are lower in the lattice or incomparable [Goguen and Meseguer, 1982; Sabelfeld and A. C. Myers, 2003]. In particular, when using noninterference to enforce privacy or confidentiality, two runs of a program that differ only in secret inputs should have identical publicly observable behavior. Noninterference is the correctness condition normally associated with information flow security (section 1.4.3).

We start by taking this traditional approach, replacing secret with "weakly consistent" and public with "strongly consistent"; in other words, we determine if any "weakly consistent" data can influence any "strongly consistent" data by comparing the possible runs of transactions. We cannot simply compare pairs of runs, however, because systems built using MixT are inherently concurrent and nondeterministic; two runs may differ simply as a result of acceptable nondeterminism. We instead consider *sets* of possible runs generated by keeping the deterministic program inputs fixed, but varying the nondeterministic choices made by the program. We say that weakly consistent data has an improper influence in this program if varying weakly consistent data introduces new strongly consistent data into the set. Put another way, varying weakly consistent data should only affect strongly consistent values in ways already permitted by the inherent nondeterminism of the system. This *possibilistic* notion of information flow is called generalized noninterference [McCullough, 1987].

Possibilistic security has been shown to be problematic in its original setting of confidentiality, because information can be leaked via refinement [Smith and Volpano, 1998; Zdancewic and A. C. Myers, 2003]. In the context of consistency and other integrity-like properties, it does not seem to be a major concern [Liu and A. C. Myers, 2014].

1.4.3 *Consistency as Information Flow*

To enforce generalized noninterference, we treat consistency as a form of information-flow integrity [Biba, 1977] and use an information-flow type system [Sabelfeld and A. C. Myers, 2003] to outlaw bad programs. Previous work [Smith and Volpano, 1998] has shown that generalized noninterference is soundly enforceable using this style of security type system. In such a type system, values are associated with a label drawn from a lattice, which in this case is a lattice of consistency levels. The strongest possible consistency is the lowest point in the lattice, denoted $\bot$, and the weakest consistency is $\top$. To enforce consistency, information should not be influenced by other information whose consistency is not at least as strong. Therefore, as in other work on information flow, legal information flow is upward in the lattice.

In the case of the buggy contest in section 1.2.3, the transaction creates a banned information flow from the inbox size (weak) to the `declare_winner()` operation (strong). In information-flow terms, this is an implicit flow [Sabelfeld and A. C. Myers, 2003]. The type system of MixT (section 1.5.2) statically catches invalid flows, whether implicit or explicit, and rejects unsafe transactions.

1.4.4 *Transaction Splitting*

We now turn to the difficult task of implementing noninterfering transactions against multiple backing databases. Consider again the message delivery code in fig. 1.4. This code is noninterferent and is therefore safe in principle, but because it involves three different consistency levels, it is nonetheless quite difficult to implement, as discussed in section 1.2.2.

MixT implements mixed-consistency transactions like this one by automatically splitting their code into a single sub-transaction per involved store. A key insight is that safe splitting is always possible because information flow restrictions prevent weakly consistent data from affecting strongly consistent data either directly or indirectly within a transaction. Hence, transactions can be split so that their stronger-consistency parts are executed earlier. This allows each sub-transaction to be safely re-executed in the case of a transaction abort, avoiding the pitfalls inherent to partitioning data across systems outlined in section 1.2.2. This splitting does not automatically preserve atomicity, the subject of section 1.4.5.

In general, a *split transaction* consists of a sequence of syntactically separate transaction phases. For each consistency level in the transaction, there is a single phase for all operations with that consistency level. MixT determines which data are communicated between phases, preserving only the information necessary to execute subsequent phases.

For example, the message delivery transaction is split into linearizable, causal, and eventual phases (in order of decreasing strength of consistency guarantees), corresponding to the consistency levels used by the transaction.

The most challenging aspect of transaction splitting is the treatment of loops, which makes splitting quite different than in previous work on automatic transaction splitting [Cheung et al., 2012; Zdancewic, Zheng, et al., 2002]. Like all expressions, each loop's condition must be evaluated within a single phase, but the body of the loop might contain statements that execute in different phases. A loop spanning multiple consistency levels, such as in the message delivery transaction, must therefore be re-executed for each consistency level.

The information-flow type system ensures that all statements that affect the loop's condition occur at the first and strongest phase in which the loop appears. In this first phase, MixT explicitly records the results of each conditional test for the loop, replaying them during the loop execution in subsequent phases. For more detail about this process and a worked example, see section 1.5.3.

1.4.5 *Whole-Transaction Atomicity*

Static transaction splitting produces a single sub-transaction per underlying store, allowing us to inherit the guarantees of isolation and atomicity provided for all operations on that store. Splitting does not, however, guarantee atomicity for the entire transaction, since commits to stronger stores happen before commits to weaker ones. To ensure atomicity, MixT programs must be prevented from observing the effects of partially committed transactions. When atomicity is guaranteed by at most one of the stores to which a transaction writes, no extra machinery is needed. However, for

the rare transaction that writes to multiple atomic stores, we introduce *witnesses*, which lock affected objects.

During each phase's execution, MixT creates a special *write witness* object for each mutation, indicating that a lock has been acquired on the object being mutated. At the end of each phase, MixT creates a single *commit witness*, a special object which indicates that all locks acquired during this transaction have been released. Only one witness is produced per transaction, but a copy of it is sent to every store on which writes were performed. If a MixT transaction encounters a write witness, it must suspend execution until it encounters the corresponding commit witness.

The witness mechanism ties together phases of split transactions across mutually unaware systems. By creating an explicit object during each transaction and blocking future progress until it has appeared, we guarantee atomicity; the full transaction will be visible to all future transactions.

The witness mechanism should impose relatively little overhead because its use should be rare. Further, several optimizations (section 1.8.1) can reduce its overhead. The performance evaluation (section 1.9) shows that a complex MixT program can achieve reasonable throughput even when witnesses are used. We revisit witnesses in more detail in section 1.5.4; formal arguments regarding the correctness of witnesses can be found in section 1.6.

$$
\begin{aligned}
(\textit{Location})\ m &::= x \mid m.x \\
(\textit{Expr})\ e &::= m \mid x_1 \oplus x_2 \mid \ominus x \mid x_0.f(x_1, \ldots, x_n) \\
(\textit{Stmt})\ s &::= \texttt{var}\ x = e\ \texttt{in}\ s \mid \texttt{remote}\ x = e\ \texttt{in}\ s \\
&\quad \mid m = x \mid \texttt{return}\ x \mid \texttt{while}\ (x)\ s \\
&\quad \mid \texttt{if}\ (x)\ s_1\ \texttt{else}\ s_2 \mid \{s_1, \ldots, s_n\}
\end{aligned}
$$

Figure 1.7: MixT flattened syntax.

1.5 FORMALIZING THE MIXT LANGUAGE

1.5.1 *Desugared Language*

To facilitate transaction splitting, MixT's surface syntax is translated to a "flattened" language whose syntax appears in fig. 1.7. Where the surface and flattened languages coincide, they share the same semantics. There are a few notable changes from the surface language. All expressions are flattened by the compiler using standard techniques [Sabry and Felleisen, 1993]. The pointer-like syntax $*e$ and e`->`x is replaced by the ability to declare `remote` variables bound to handles. Semantically, `remote` variables directly correspond to the referenced location on an underlying store. Updates to these variables are reflected at the store, and uses of these variables query the store directly for their value. Unlike in the surface language, both `var` and `remote` introduce explicit scopes for their bindings.

$$\frac{\Gamma \vdash e : \tau \qquad \begin{array}{c}\Delta \mid \Gamma \mid pc \vdash e : \ell \\ \Delta, x_V : \ell \mid \Gamma, x_V : \tau \mid pc \vdash s : \ell_2\end{array} \qquad x \notin \Gamma \qquad x \notin \Delta}{\Delta \mid \Gamma \mid pc \vdash \texttt{var}\ x\ \texttt{=}\ e\ \texttt{in}\ s : \ell}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{Handle}\langle \tau, \ell_1 \rangle \qquad \Delta \mid \Gamma \mid pc \vdash e : \ell_2 \\ \Delta, x_R : \ell \mid \Gamma, x_R : \tau \mid pc \vdash s : \ell' \qquad \ell_1 \sqcup \ell_2 \sqsubseteq \ell\end{array} \qquad x \notin \Gamma \qquad x \notin \Delta}{\Delta \mid \Gamma \mid pc \vdash \texttt{remote}\ x\ \texttt{=}\ e\ \texttt{in}\ s : \ell}$$

$$\frac{\Delta, x\_ : \ell \mid \Gamma \mid pc \vdash e : \ell}{\Delta, x\_ : \ell \mid \Gamma \mid pc \vdash x\ \texttt{=}\ e : \ell} \qquad \textsc{remote-read}\ \frac{pc \sqsubseteq \ell}{\Delta, x_R : \ell \mid \Gamma \mid pc \vdash x : \ell} \qquad \Delta, x_V : \ell \mid \Gamma \mid pc \vdash x : \ell$$

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell' \qquad \ell' \sqsubseteq \ell}{\Delta \mid \Gamma \mid pc \vdash e : \ell} \qquad \frac{\Delta \mid \Gamma \mid pc \vdash e : \ell \qquad \Delta \mid \Gamma \mid pc \sqcup \ell \vdash s : \ell'}{\Delta \mid \Gamma \mid pc \vdash \texttt{while}\ (e)\ s : \ell}$$

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell \qquad \Delta \mid \Gamma \mid pc \sqcup \ell \vdash s_1 : \ell' \qquad \Delta \mid \Gamma \mid pc \sqcup \ell \vdash s_2 : \ell'}{\Delta \mid \Gamma \mid pc \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 : \ell}$$

$$\frac{pc \sqsubseteq \bot \qquad \Delta \mid \Gamma \mid pc \vdash e : \ell}{\Delta \mid \Gamma \mid pc \vdash \texttt{return}\ (e) : \top}$$

Figure 1.8: Selected consistency typing rules for MixT. The labeled REMOTE-READ rule is unusual. Also unusually, statements have explicit labels; these are used to determine the phase in which the statement should run during transaction splitting.

1.5.2 *Statically Checking Consistency Labels*

Consistency is enforced in MixT by statically checking information flow using a largely standard type system for static information flow [Sabelfeld and A. C. Myers, 2003].

Figure 1.8 gives selected consistency typing rules for the language. Ordinary rules for typing judgments $\Gamma \vdash e : \tau$ are not presented because they directly use the C++ type system; the presented rules are only for *consistency judgments* $\Delta \mid \Gamma \mid pc \vdash e : \ell$. Environments Δ and Γ keep track of the labels and types of variables, respectively, with local and remote variables distinguished lexically by subscripts V and R. The label pc (for *program counter*) bounds the consistency of control flow.

The rules assign each statement and expression a consistency label ℓ that reflects the weakest consistency of any information used to compute it. The label on statements, used during transaction splitting, is derived directly from subexpressions and is unaffected by substatements. During static checking, consistency originates from the consistency labels on handles, which derive from their stores. Variables captured from the environment outside of the transaction are labeled with the strongest ($\bot$) consistency; all other labels are automatically inferred from the transaction code.

One non-standard aspect of the rules is that all accesses to `remote`-bound variables are treated as effectful, requiring the same pc consistency to read a `remote` location as to write it (REMOTE-READ). This restriction is imposed for correct transaction splitting, to enforce the necessary condition that all remote operations at a single consistency level execute before any remote operations at a weaker level.

$$
\begin{aligned}
(\textit{Expr})\ e &::= \ldots \mid \texttt{rand()} \mid \texttt{peek}(x) \\
(\textit{Stmt})\ s &::= \ldots \mid \texttt{release\_all}(n) \\
&\quad \mid \texttt{acquire}(x, n, \ell_1, \ldots, \ell_n) \\
&\quad \mid \texttt{advance}(x) \mid \texttt{advance remote}(x) \\
(\textit{Phase})\ p &::= s_\ell \\
(\textit{Transaction})\ t &::= \mathcal{T}\{p_1; p_2; \ldots; p_n\}
\end{aligned}
$$

$$[\![\texttt{remote}\ x = e\ \texttt{in}\ s : \ell]\!]_\ell \triangleq \texttt{remote}\ x = e\ \texttt{in}\ [\![s]\!]_\ell$$

$$\frac{\ell \not\sqsubseteq \ell'}{[\![\texttt{remote}\ x = e\ \texttt{in}\ s : \ell]\!]_{\ell'} \triangleq [\![s]\!]_{\ell'}}$$

$$\frac{\ell \sqsubseteq \ell'}{[\![\texttt{remote}\ x = e\ \texttt{in}\ s : \ell]\!]_{\ell'} \triangleq \{\texttt{advance binding}(x), [\![s]\!]_{\ell'}\}}$$

$$[\![\texttt{while}\ (e)\ stmt : \ell]\!]_\ell \triangleq \texttt{while}\ (e)\ [\![stmt]\!]_\ell \qquad \frac{\ell \not\sqsubseteq \ell'}{[\![\texttt{while}\ (e)\ stmt : \ell]\!]_{\ell'} \triangleq \{\}}$$

$$\frac{\ell \neq \ell' \qquad \ell \sqsubseteq \ell'}{[\![x = e : \ell]\!]_{\ell'} \triangleq \texttt{advance}(x)} \qquad \frac{\ell \neq \ell' \qquad \ell \sqsubseteq \ell'}{[\![x : \ell]\!]_{\ell'} \triangleq \texttt{peek}(x)} \qquad [\![x : \ell]\!]_\ell \triangleq x$$

$$\frac{L = \{\ell_1, \ell_2, \ldots, \ell_n\}}{\mathcal{S}[\![stmt]\!]_L \triangleq \mathcal{T}\{[\![stmt]\!]_{\ell_1}; [\![stmt]\!]_{\ell_2}; \ldots; [\![stmt]\!]_{\ell_n}\}}$$

Figure 1.9: Selected transaction splitting rules.

Omitted from fig. 1.8 is the rule governing endorsement, discussed in section 1.7.

1.5.3 *Transaction Splitting*

Fig. 1.9 gives selected rules for splitting transactions into phases based on consistency labels. The translation $[\![\cdot]\!]_\ell$ generates the code for the transaction phase at consistency level ℓ.

```
var iterator = users, // Phase: strict serializability
var loopindex = iterator.isValid(),
while (loopindex) {
  loopindex = iterator.isValid(),
  var temporary0 = iterator->v,
  iterator = iterator->next
}
```

```
advance(loopindex), // Phase: causal consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary0),
  var temporary1 = peek(temporary0).inbox.insert(post)
}
```

```
advance(loopindex), // Phase: eventual consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary1),
  logger.log(peek(temporary1))
}
```

Figure 1.10: The message delivery transaction after splitting, lifted back to the surface syntax.

Recall that each statement in the flattened language is associated with exactly one consistency level. Intuitively, transaction splitting preserves a statement in phase ℓ when the statement's label matches ℓ and otherwise omits it from the phase. However, statements associated with a nested scope, such as `while` and `var`, may execute their contained statements in a different phase.

Once some variable x is introduced in a phase, it may be used—but not assigned—in any later phase. During the phase in which x is introduced, every binding and assignment to x is stored in an implicit iterator. During subsequent phases, this iterator is used to replay x's previous values. In these subsequent phases, uses of x are replaced with `peek(`x`)` expressions, which return the current value of x's implicit iterator. Mutations to x are replaced with `advance(`x`)`, which advances x's implicit iterator. Remote-

bound variables require the additional `advance binding(`x`)` construct. Recall that the `remote` x `=` e construct binds x as an alias to the remote state described by e. If this binding appears in a loop, then the value of e may shift, causing x to be bound to a series of remote locations. The `advance binding(`x`)` statement cycles through these, while the `advance(`x`)` statement cycles through assignments to the remote object itself. These and other syntactic extensions required by transaction splitting are shown in fig. 1.9.

For example, the split transaction generated from the message delivery transaction contains three non-local phases, one for each distinct consistency level used, as shown in fig. 1.10 (for clarity, the code is presented in the surface syntax). Updates to the freshly generated variables `loopindex`, `temporary1`, and `temporary0` are logged; the expression `peek(`x`)` accesses an iterator over the previous values of x, and the expression `advance(`x`)` advances this iterator. Neither of the original variables `iterator` and `users` is necessary for any subsequent phase, so they are discarded upon successful commit of the first phase.

1.5.4 *Enforcing Atomicity in Split Transactions*

After transaction splitting, the MixT compiler augments the split transaction with code to create witnesses. First, the transaction generates a new variable `wit` (as shown in the GENERATE rule of fig. 1.11) containing the uniquely-generated name of our commit witness[2]. Next, it inserts the statement `acquire(`x`,wit,`*phases*`)` before all mutative operations (ACQUIRE

2 Our implementation handles name generation by selecting a random 63-bit number; these names are short-lived and can be garbage collected within minutes, keeping the probability of collision orders of magnitude below the probability of catastrophic hardware failure.

GENERATE

$$\frac{p_i, p_j \in \mathcal{P}.p_i \neq p_j \wedge writes(p_i) \wedge writes(p_j) \qquad \mathcal{P} = [p_1, \ldots, p_n]}{[\![[p_1, \ldots, p_n]]\!] \triangleq [\texttt{var wit = gensym()}_{\perp}, [\![p_1]\!]_2, \ldots, [\![p_n]\!]_2]}$$

RELEASE

$$[\![phase_n]\!]_2 \triangleq [\![phase_n]\!]_3; \texttt{release\_all(wit)}$$

ACQUIRE1

$$\frac{\texttt{x} \text{ is remote-bound}}{[\![\texttt{x = e}]\!]_3 \triangleq \texttt{x=e; acquire(x,wit,}\mathcal{P}\texttt{)}}$$

ACQUIRE2

$$[\![\texttt{x.f(...)}]\!]_3 \triangleq \texttt{x.f(...); acquire(x,wit,}\mathcal{P}\texttt{)}$$

Figure 1.11: Selected rules for witness modification. $\mathcal{P}$ is the list of transaction phases generated during splitting of a MixT transaction, and *writes* is a boolean function indicating whether a write is performed in a phase. Semicolon separates instructions.

rules). This statement creates a write witness, writing it alongside x. At the end of the phase, the compiler appends `release_all(wit)` (RELEASE rule) , which writes the commit witness itself. Witnesses are only required for transactions which perform writes to remote storage in multiple phases (captured in the precondition to the GENERATE rule). Witnesses are required to achieve the mixed-consistency property of crefsec:mixed-consistency-def for non-compositional models, and are essential to atomicity; however, they may be manually disabled. If witnesses are disabled, then MixT can guarantee mixed-consistency for compositional models (and the compositional fragments of non-compositional models), and supplies at most the READ COMMITTED isolation level.

As an example, we present witness generation for the familiar message delivery transaction in fig. 1.1. A new first phase contains witness generation; in all phases, each mutative operation is immediately followed by a

```
var wit = rand()
```

```
var iterator = users, // Phase: linearizable
var loopindex = iterator.isValid(),
while (loopindex) {
  loopindex = iterator.isValid(),
  var temporary0 = iterator->v,
  acquire(temporary0,wit,linearizable,causal,eventual),
  iterator = iterator->next
}
release_all(wit)
```

```
advance(loopindex), //Phase: causal consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary0),
  var temporary1 = peek(temporary0).inbox.insert(post),
  acquire(temporary1,wit,linearizable,causal,eventual)
}
release_all(wit)
```

```
advance(loopindex), //Phase: eventual consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary1),
  logger.log(peek(temporary1))
}
release_all(wit)
```

Figure 1.12: The message delivery transaction after witness insertion, lifted back to the surface syntax

call to `acquire`. At the end of each phase, MixT copies the commit witness to the store via `release_all`.

The purpose of witness transformation is simple: each call to `acquire(`x`, wit, ...)` acquires a logical lock on x, which is released by the corresponding call to `release_all(wit)`. Any transaction which observes the "lock acquire" event (`acquire`) must now wait for the corresponding "lock release" event (`release_all`) before proceeding at each participating phase.

Witnesses are implemented as simple objects stored directly on remote stores. Write witnesses contain the locations of all corresponding commit witnesses and a list of all consistency levels involved in the transaction, while commit witnesses are empty objects.

Additional run-time checks beyond those inserted by fig. 1.11 are required to check for witnesses. At run time, whenever MixT attempts to read a `remote`-bound value, it also reads that value's potential witness location, checking for a write witness. If it finds one, it adds the discovered commit witness's location to a *witness worklist* for each phase listed in the write witness.

Before MixT begins executing a phase p, it first iterates through p's witness worklist. For each commit witness w in the worklist, MixT polls p's store until w becomes available, and then removes w from the worklist. As described in section 1.8.2, this polling loop occurs on the remote store itself. Once the commit witness has appeared on all replicas, it may be safely removed from the store; the specifics of this process are found in section 1.8.1.

These additional run-time checks prevent a transaction from observing its own causal past; if a certain transaction phase has witnessed a past

```
class DataStore<Label> {
   ...
   bool exists(Name name);
   Handle<...> existingObject(Name name);
   Handle<...> newObject(Name name);
};
```

Figure 1.13: Enhanced DataStore interface for witnesses.

transaction, then we must ensure that all future transaction phases can also witness that transaction.

To support witnesses, we extend the requirements on underlying storage to include a key–value API through which witnesses can be named (fig. 1.13). These APIs take the form of type constraints; the return and parameter types must support certain operations, but do not need to be a precise MixT-specified type.

Note that using witnesses does not cause the resulting transaction to become fully serializable; if a weak consistency level allows stale reads, then stale reads can still occur during the weak phase of a mixed-consistency transaction. A mixed transaction might fail to read values committed by a previous fully weak transaction. Even with witnesses, the code in fig. 1.2 remains incorrect. Also, atomicity is only guaranteed when the underlying store provides it; consistency levels which do not have a useful notion of atomicity—for example, eventual consistency—therefore do not employ write or commit witnesses.

1.6 CORRECTNESS

1.6.1 *Mixed Isolation*

To argue correctness of mixed-consistency transactions, we must first establish a notion of *mixed isolation* levels. As with consistency models, we represent isolation levels as trace properties. To have a common framework in which to discuss both consistency and isolation, we associate each consistency level ℓ with an isolation level $\mathcal{I}_\ell$. We again characterize the behavior of the program as a set of admitted traces T, where each trace $t \in T$ is a sequence of events e. An event e is now a 6-tuple $(a, o, \ell, \mathcal{T}, v, S)$; members of this tuple are as defined in section 1.4.1 with the exception of $\mathcal{T}$, a unique token corresponding to the transaction to which this event belongs. We lift $\lfloor$ to this new setting as $\lfloor_{\mathcal{I}}$. We define $\mathcal{T}_e = \pi_4(e)$ as the transaction of an event. Our traces now include explicit events for transaction begin ($begin_{\mathcal{T}}$), abort ($abort_{\mathcal{T}}$), and commit ($commit_{\mathcal{T}}$), along with corresponding actions *begin*, *abort*, and *commit*. Similarly to consistency, the meaning of each isolation level $\mathcal{I}_\ell$ is given by its isolation model I_ℓ, an (infinite) set of traces containing all executions which satisfy the guarantees of the isolation level. We use $<$ as the order for events within traces.

We define a forgetful function $F(t) = \{(a, o, \ell, v, S) \mid (a, o, \ell, \mathcal{T}, v, S) \in t \wedge a \notin \{end, commit, abort\}\}$ which drops transaction events from a trace, converting it back to section 1.4.1's notion of application traces. We say that an isolation model I_ℓ is well-formed if $\{F(t) \mid t \in I_\ell\} \subseteq \ell$; that is,

normal events in the isolation model are also in its associated consistency model. We now define *mixed isolation* as an analog of *mixed consistency*:

Definition 1.6.1 (Mixed isolation of trace t)**.**

$$\forall \ell, t \downarrow_{\mathcal{I}} \ell \in I_\ell$$

We now proceed to argue that our mechanisms of transaction splitting and witnesses are sufficient to guarantee both mixed consistency and atomicity for mixed isolation.

1.6.2 *Mixed Consistency and Noninterference through Splitting*

We first argue that the trace generated by any sequence of transactions which do not utilize the `return` or `endorse` constructs are guaranteed to preserve mixed-consistency. This argument is naturally dependent on the nature of the consistency model in question. If the consistency model, as is traditional, includes only information about independent reads and writes made to individual memory locations then it is easy to see that MixT satisfies mixed consistency. Because MixT associates each data object or memory location with a single consistency model, all reads of some value from a memory location at some consistency level ℓ must be paired with a write to that location at level ℓ. Thus, for all reads in a trace, the projection operator ($\downarrow$) also preserves all matching writes in that trace. In this sort of consistency model, the projection operator simply selects sets of memory locations.

We next turn to enforcing noninterference (again, for transactions without `endorse` or `return`). This follows directly from our splitting algorithm

and type system. In MixT, labels are derived only from remote actions; during transaction splitting, these remote actions are segregated into phases. For any pair of labels $\ell_w \sqsubset \ell_s$, all ℓ_s-labeled operations occur before ℓ_w's phase, and thus ℓ_w actions cannot interfere with ℓ_s actions. For pairs of incomparable labels ℓ_l and ℓ_r, our compiler chooses a deterministic order in which to execute their phases, and our runtime executes both phases with the environment of their last ancestor. Thus even in this case, our phases cannot interfere. The remote actions themselves are expected to be carried out on distinct, mutually-unaware backing storage systems; these systems are strongly isolated from each other, which prohibits any additional influence channels.

We make no argument that the trace of an entire program will adhere to our properties; as MixT is embedded within the context of a larger C++ program, it will always be possible for programmers to take the value returned by one transaction, ignore its consistency, and use it inappropriately in a subsequent transaction. Further, it is unclear what semantics consistency labels should have outside of a transaction; the moment a transaction ends, all its observations may be invalidated by a subsequently-scheduled transaction, rendering even "strongly-consistent" labeled values untrustworthy. A program in which all interactions with the database are made via void-returning MixT transactions *will* enjoy our correctness properties; in this case, the surrounding C++ code is isolated from the MixT transactions.

1.6.3 *Atomicity through Write Witnesses*

We now address the claim that our split transactions preserve atomicity, given underlying stores which themselves preserve atomicity. As the strongest purely atomicity-based property granted by a standard isolation level is `read committed` [Adya, 1999], we limit ourselves to demonstrating that write witnesses provide the guarantees of the read committed isolation level when all participating stores also provide the guarantees of the read committed isolation level.

We first observe that transactions which perform writes in a single phase trivially satisfy read committed, as their underlying stores provide this guarantee. We therefore consider an arbitrary sequence of writes $w_1 < \cdots < w_n$ carried out by the same transaction $\mathcal{T}_w$, but executed against distinct underlying stores. Let $\mathcal{T}_r$ be some transaction which reads some $w_n \in w_1, \ldots, w_{n-1}$. This $\mathcal{T}_r$ must therefore encounter a write witness for w_n, and will check for the corresponding commit witness at all participating stores before proceeding. The slowest sub-transaction of $\mathcal{T}_w$, upon commit, will write the final required commit witness for w_n. As this witness is only written when the final commit of $\mathcal{T}_w$ occurs, we conclude that $\mathcal{T}_r$'s read itself will occur after $\mathcal{T}_w$'s commit. Thus, atomicity is preserved.

1.6.4 *Read Witnesses*

The witness mechanism as described to this point only associates witnesses with writes and commits, guaranteeing full atomicity but not full isolation; this mechanism cannot always enforce isolation stronger than

`read committed` [Berenson et al., 1995]. Transactions which operate over snapshot-isolating ([Berenson et al., 1995]) levels ℓ_1, ℓ_2, and ℓ_3, perform reads at ℓ_1 and ℓ_2, and then use those reads in level ℓ_3 may see violations of snapshot-isolation at ℓ_3—despite all of ℓ_1, ℓ_2, and ℓ_3 guaranteeing snapshot isolation. If ℓ_1 permits linearizable reads, then one can recover snapshot isolation with *read witnesses*, by inserting calls to `acquire` after every read and checking for read/write witnesses before each write. We have observed few cases in which read witnesses are required; indeed none of the motivating examples for this work require them. We have not observed compelling examples in which read witnesses are impossible, as it would require multiple distinct consistency levels that are snapshot-isolated, none of which support linearizable reads. Read witnesses are currently only partially implemented in the MixT compiler.

1.6.5 *Mixed Isolation through Splitting and Witnesses*

We return to the argument in section 1.6.3. Observe that simply replaying its argument with read witnesses and reads, rather than write witnesses and writes, produces a satisfying argument for isolation; any potential violation of isolation must involve a write w_{new} which occurs during some transaction which has already read w_{old}; but as this transaction acquired a read witness on w_{old}, the write to w_{new} must be delayed until transaction commits.

We now are ready to demonstrate mixed isolation. We observe that any violation of mixed isolation must be due to a failure of mixed consistency, atomicity, or isolation. We have argued that witnesses and transaction split-

ting are sufficient to protect against failures of mixed consistency, atomicity, and isolation independently; we now conclude that their conjunction is sufficient to grant mixed isolation.

1.6.6 *Mixed Consistency and Compositionality*

Those familiar with reasoning about consistency models may wonder how MixT can ensure mixed consistency (Defn. 1.4.2) between non-compositional consistency models [Herlihy and Wing, 1988]. In general, non-compositional consistency models fail to capture client-centric notions of dependency between events from distinct systems; as no store knows the full set of actions performed by a client, no store is in a position to restrict possible orderings across all actions [Protic, Tomasevic, and Milutinovic, 1996]. It is clearly unrealistic to ask stores to track events of which they are intentionally unaware. We therefore use a lightweight cross-store tracking mechanism to explicitly capture session order and causality at points where execution switches between stores.

Witnesses are the core of our cross-store tracking mechanism. Whenever a client performs a *cross-store sequence* of transactions (t_1, w_2) in which some stores referenced by w_2 aren't used by t_1 (or vice-versa), if those stores provide non-compositional consistency models and w_2 performs writes to them, then we must enhance w_2. Specifically, we extend w_2 with extra phases corresponding to non-compositional models in t_1 which did not already appear in w_2. These new phases exist only to write a commit witness for t_1.

Any client which attempts to read w_2's writes will find these witnesses, and thus any subsequent read at any of t_1's stores will block until the matching commit witness is available. This conveys the session-order relationship between t_1 and w_2 to all stores; effectively, the commit witness causes the stores of t_1 to communicate the client's session order to the rest of the system.

This mechanism is sufficient to achieve the mixed-consistency property defined in section 1.4.1. We show this by contradiction. If it were not sufficient, there would be an execution trace t and consistency level ℓ such that $t \downharpoonright \ell \notin T_\ell$. Data annotated at ℓ lives on a single store which guarantees ℓ; as t does not guarantee ℓ, we conclude t is a mixed trace. Furthermore, as all stores store disjoint sets of objects, any inconsistency in t must arrive from a violation of session order or observation order—there must be some sequence of events $e_w, e_{\ell'}, e_r \in t$ in which e_w and e_r live on the store of ℓ, $e_{\ell'}$ lives on the store of some other level ℓ', and e_r is erroneously sequentialized before e_w. But e_w and $e_{\ell'}$ are in a cross-store sequence, so $e_{\ell'}$ was accompanied by a commit witness at e_w's store. Therefore, the session executing e_r must have previously read this commit witness, forcing e_r to sequentialize after e_w.

1.7 UPGRADING CONSISTENCY

1.7.1 *Semantics and Motivation*

As described up to this point, MixT transactions maintain a strict separation between consistency levels; it is impossible to use a weakly consistent

```
  if ((a.inbox.strong_read().size() >= 1000000 &&
      b.inbox.strong_read().size() < 1000000)
       .endorse(strong)) {
    a.declare_winner()
} else
  if ((a.inbox.strong_read().size() < 1000000 &&
      b.inbox.strong_read().size() >= 1000000)
       .endorse(strong)) {
    b.declare_winner()
}
```

Figure 1.14: Strongly consistent contest logic with endorsement.

value to influence a strongly consistent operation. This rigid separation provides strong semantic guarantees, but can be limiting.

As an example, consider the mail-delivery example from fig. 1.2. During the contest, mail is delivered with causal consistency; users who view their inboxes may see a slightly stale view of the mail they've received—an acceptable semantics given the already variable latency of mail delivery itself. The system, however, needs to perform a strongly consistent read to determine a winner once the contest has finished. Let's imagine that the store supports such an operation, and exposes it to MixT via the name `strong_read`.

In MixT, the result of any expression can be declared to have an arbitrary consistency via the built-in operation `endorse(`*label*`)`. It behaves as a type-cast; the MixT compiler makes no effort to ensure that endorsements are used appropriately, as their validity often depends on complex system properties not visible to the compiler. Like all of weak consistency, endorsements must be used with care.

Using our hypothetical `strong_read` operation and MixT's `endorse` keyword, we can fix our transaction (fig. 1.14). This code is correct, but runs all operations at a strong consistency level—exactly what we were trying

```
if (((a.inbox.size() >= 1000000 &&
     b.inbox.size() < 1000000)
 && (a.inbox.strong_read().size() >= 1000000 &&
     b.inbox.strong_read().size() < 1000000))
      .endorse(strong)) {
   a.declare_winner()
} else
if (((a.inbox.size() < 1000000 &&
     b.inbox.size() >= 1000000)
 && (a.inbox.strong_read().size() < 1000000 &&
     b.inbox.strong_read().size() >= 1000000))
      .endorse(strong)){
   b.declare_winner()
 }
```

Figure 1.15: Efficient contest logic with endorsement. The programmer has introduced an additional check involving a rare strong read for when the contest is believed to be over. We also must endorse the enclosing conditional, as it still creates an indirect flow to `declare_winner`.

to avoid. To improve performance, we can guard each strongly consistent read with a preliminary weakly consistent test, restoring causal execution for most transactions while ensuring the declaration of a winner is still guarded by a strongly consistent condition (fig. 1.15). The resulting consistency of the strong `declare_winner()` operation is no longer separable from the causally consistent read; we have accepted the possibility that our winner may be declared late.

1.7.2 *Compiling Endorsements*

Because of transaction splitting, endorsement is not straightforward. It fundamentally involves running weaker commands before stronger ones — and thus requires more than one phase per underlying store. But naively adding additional phases will not provide acceptable semantics; having one phase per underlying store is key to MixT's isolation guarantees.

$\ell \in$ **Label**
$e ::= \ldots \mid e.\texttt{endorse}(\ell) \mid e.\texttt{ensure}(\ell)$

Figure 1.16: Syntax extensions for endorsement.

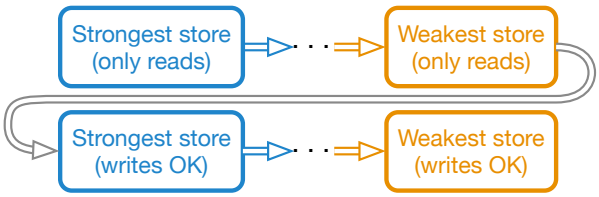


Figure 1.17: Transaction phases with endorsement.

We address these concerns with two mechanisms: *read-only phases* and read witnesses. As in other information-flow languages, endorsement is indicated by annotating the endorsed expression(fig. 1.16). The compiler separates the transaction into two parts: the pre-endorse part and the post-endorse part. To ensure atomicity, the pre-endorse part of the transaction is checked to ensure that it contains no writes. The code is then split into phases in the usual way, except that an artificial "pre-endorse label" is first joined with all labels in the pre-endorse code so that pre-endorse code appears to the compiler to have stronger consistency than all post-endorse code. This process is depicted in Figure 1.17. If full isolation is required, read witnesses (section 1.6.4) may be employed. An optimization which uses read validation in place of read witnesses (as in optimistic concurrency control) is also possible [Kung and Robinson, 1981].

1.8 IMPLEMENTATION

MixT has been implemented as four separate C++17 components, numbering almost 30,000 lines in total: the transactions language compiler (10k), the core library (2.8k), the tracking mechanism (1k), the Postgres implementation (1.4k), and support utilities (14k). The compiler can be further broken down into various phases: parsing (1.4k), A-normal transformation (500), type inference (1k), label inference (1.7k), endorsement (250), splitting (1.6k), optimization (1k), and codegen (1.4k). The current implementation supports an unbounded number of backing stores in a single application.

To evaluate MixT, we also developed several sample backing stores, operating either in-memory or based on PostgreSQL 9.4. These interfaces expose a selected set of prepared statements as custom operations and are designed to provide linearizability (with strict serializability), causal consistency (with snapshot isolation), and eventual consistency (with read-uncommitted isolation.)

1.8.1 *Efficiency Optimizations*

Recall that mixed-consistency transactions use witnesses to ensure atomicity, wherein an extra read operation accompanies each stated read, and an extra write operation accompanies each transaction. As described so far, this mechanism would frequently encounter stale values, harming performance for no gain in safety. Additionally, commit witnesses would slowly accumulate on the store, wasting storage. Ideally, we would be

able to determine whether a value was stable across an entire store, and therefore did not require such defensive tracking behavior. To accomplish this we observe that, in order to maintain its own guarantees regarding operation order, our non-compositional store likely has some notion of a timestamp or version number already available for internal use. This assumption proves true in practice: systems such as COPS, Eiger, 2-master PostgreSQL, Bolt-on, TARDIS, HBase, and Mongo (via Vermongo) all either use these vector clocks directly or are easily modified to employ them [Bailis, Ghodsi, et al., 2013; Crooks et al., 2016; Lloyd et al., 2011, 2013; Plugge, Membrey, and Hawkins, 2010]. We enhanced our `Handle` API to allow client stores to expose this notion of time to MixT through an optional `timestamp` method . We augment read and write witnesses to include the "current time" of transaction commit, and provide a lightweight TCP protocol through which backing stores can notify MixT clients of the most recent version number which is guaranteed to have reached the entire store. If the ability to access an accurate transaction commit time from within a transaction does not exist, commit witnesses can be augmented to include the addresses of all read and write witnesses created during the transaction. Leveraging this additional information, MixT avoids generating witnesses when the objects involved are already widely available, and can safely remove stale commit witnesses.

1.8.2 *Remote Execution in MixT*

With MixT's ability to split transactions into phases comes the opportunity to distribute the transaction code itself. By deploying a lightweight worker

process alongside existing backing stores, MixT application programmers can run transaction phases directly at stores, incurring only a single round trip to establish each phase and collect its results – and allowing all witness checks to be carried out locally. In fact, this decision—to ship transaction code directly to the storage system—has become increasingly popular among high-performance data storage systems and is central to some modern databases [Dubey et al., 2016; Duggan et al., 2015; Kallman et al., 2008; Stonebraker and Weisberg, 2013]. MixT's approach to remote execution is straightforward. We assume that each application manages its own lightweight worker at the storage location; we leave the task of ensuring code is up-to-date to the MixT application programmer.

1.8.3 *MixT Compiler Implementation*

We implemented MixT as a domain-specific language embedded into modern C++. MixT is written in pure C++17, and can be compiled using any C++17-compliant compiler[3]. Our entire compiler is written in constexpr C++ [Meyers, 2014], allowing it to run during the "template expansion" step of compilation of the surrounding C++ code. Specifically, `mixt_`⟨*keyword*⟩ macros convert their arguments into a compile-time string, which is then parsed and compiled by our compiler. In order to link names within the transactions language to native C++ objects, the macros capture both the type of their arguments and their string representations, using these during the transaction compilation. All compilation, including transaction splitting, is accomplished alongside C++ compilation; none

3 The MixT compiler is tested under ≥clang-3.9 and ≥g++-7.1; certain syntax extensions require -fconcepts

is deferred until run time. Transactions are compiled to a set of inlined, statically bound functions which are invoked from a single point in code, allowing the C++ compiler to optimize away all function-call overhead, producing machine code quite close to the syntax specified by the transaction. This approach allows MixT to support arbitrary syntax, semantics, and type systems, without requiring an external compiler or preprocessor, and without adding unnecessary run-time overhead. We are not bound to the syntax, semantics or keywords of C++; MixT's similarity to C++ is a conscious design choice. We follow the language-as-a-library paradigm: as MixT effectively adds extra phases to existing C++ compilation, to use MixT in existing C++ projects all one must do is `#include` the MixT header files.

1.9 EVALUATION

We use the MixT implementation to model an intended application domain: user-facing application servers that share one linearizable and one causally consistent underlying storage system, where application servers are geographically close to only the causal replica they are using. We believe this closely mirrors reality. Weakly consistent storage servers can be relatively close to application servers because they are able to withstand high latencies during replication and can therefore be separated geographically; linearizable data stores are typically housed within a single data center because latencies encountered during replication have an outsized impact on overall performance.

In this setting, we explore several key questions regarding the performance of MixT:

- Do mixed-consistency transactions, as promised, offer better performance than running similarly atomic transactions with strong consistency?
- What overhead is added by the witness mechanism used to preserve consistency guarantees when non-compositional consistency levels are combined?
- On what workloads does this mechanism work well? What is the performance impact of different mixtures of mixed and pure transactions?

1.9.1 *Experimental Setup*

To measure the performance of MixT, we simulate a geo-replicated application. In our setup, logically separate application servers each maintain connections to causally consistent and serializable databases. Connections to the serializable database experience a round-trip latency of 85ms $\pm$ 10ms; connections to causally consistent databases experience a round-trip latency of 1ms. Latency to the causal system was set by measuring ping times between an Internet2-connected university and its nearest data center; latency to the linearizable system was set by measuring ping times between Internet2-connected universities on the east and west coasts of the United States. All latency simulations are provided by the netem kernel module on Linux 3.17. We employ three separate physical machines:

one hosting all clients, one hosting the causal store, and one hosting the linearizable store.

For driving load to application servers, we adopted a semi-open world model, with delay between events following an exponential distribution. We increase load by increasing the number of MixT clients, not by increasing the rate of events issued by each client.

USING POSTGRESQL AS A BACKING STORE Our causal and linearizable stores are both backed by PostgreSQL 9.4 running on dedicated machines. These instances are configured with a maximum of 2010 connections, 128 MB of shared buffers, and with both `fsync` and `full_page_writes` disabled to improve performance; the rest of PostgreSQL's configuration parameters are left at their default values.

These PostgreSQL instances consist of only two tables; one table associates integral values with integral keys and version numbers; the other table associates binary blobs with integral keys and version numbers. Any integral type is mapped to a row in the first table; all other types are mapped to a row in the second table. SQL queries over these tables are naive updates, selects, and increments (for the integral table).

Because we use PostgreSQL as a key–value store, SQL-specific performance concerns such as query optimization or parse time should not significantly affect our results.

When running as a linearizable store, PostgreSQL is put in a "normal" operating mode with a single master per object and the `SERIALIZABLE` transaction isolation level. The coding overhead required to create this interface was surprisingly small; about 180 lines of C++ code, mostly for registering prepared statements.

To configure PostgreSQL as a causally consistent store, we created four replicas of data, and partitioned client programs among the four copies. Each instance runs transactions with snapshot isolation enforcing the guarantees of causal consistency. To order operations occurring at distinct replicas, we use a vector clock as a per-row version number. Vector clock entries are just the microsecond-resolution time at each master, so vector clock maintenance does not add serialization conflicts.

A stored PL/pgSQL operation updates these version numbers upon row modification.

These mechanisms were implemented in 1,000 lines of C++ and about 100 lines of PL/pgSQL.

Snapshot Isolation enforces the guarantees of causal consistency because within a single session, all reads will reflect data no older than the previous transactions' reads, and each transaction can see the modifications made by all previous transactions from this session. When reading data, a custom PL/pgSQL stored procedure automatically merges all four replicas and produces the appropriate vector clock.

1.9.2 *Benchmarks*

We could find no existing benchmarks for mixed-consistency transactional systems. Instead, we developed two new benchmarks intended to represent the emerging mixed-consistency landscape. The first is a simple microbenchmark based on incrementing integral counters. It features read-only transactions that fetch the value at a counter, and read-write transactions which increment that value. Objects referenced during read

operations are selected from a Zipf distribution over 400,000 names; objects referenced during write operations are selected from a uniform distribution over the same names. These objects are duplicated on both a linearizable and casual store. In this benchmark, clients randomly move between causal mode where all transactions are causally-consistent, and a linearizable mode where all transactions are linearizable, with a fixed probability. We extend this benchmark to involve mixed-consistency transactions in the next section.

The second benchmark is the Message Groups example discussed in section 1.2.2. This benchmark features four more-complex transactions: message delivery (fig. 1.4), user creation, inbox checking, and group joining. User creation and inbox checking are causally consistent, while message posting and group joining are mixed-consistency transactions. Each client is assigned a range of inboxes and groups from which it selects uniformly at random.

1.9.3 *Counter Results*

The counters benchmark offers several tuning parameters for exploring the space of workloads. As copies of our complete set of objects exist on both a causal and linearizable store, we can fine-tune both the mixture of causal and linearizable operations and the combination of reads and writes in our tests.

SPEEDUP RELATIVE TO LINEARIZABILITY The most important question for MixT performance is whether mixed-consistency transactions offer

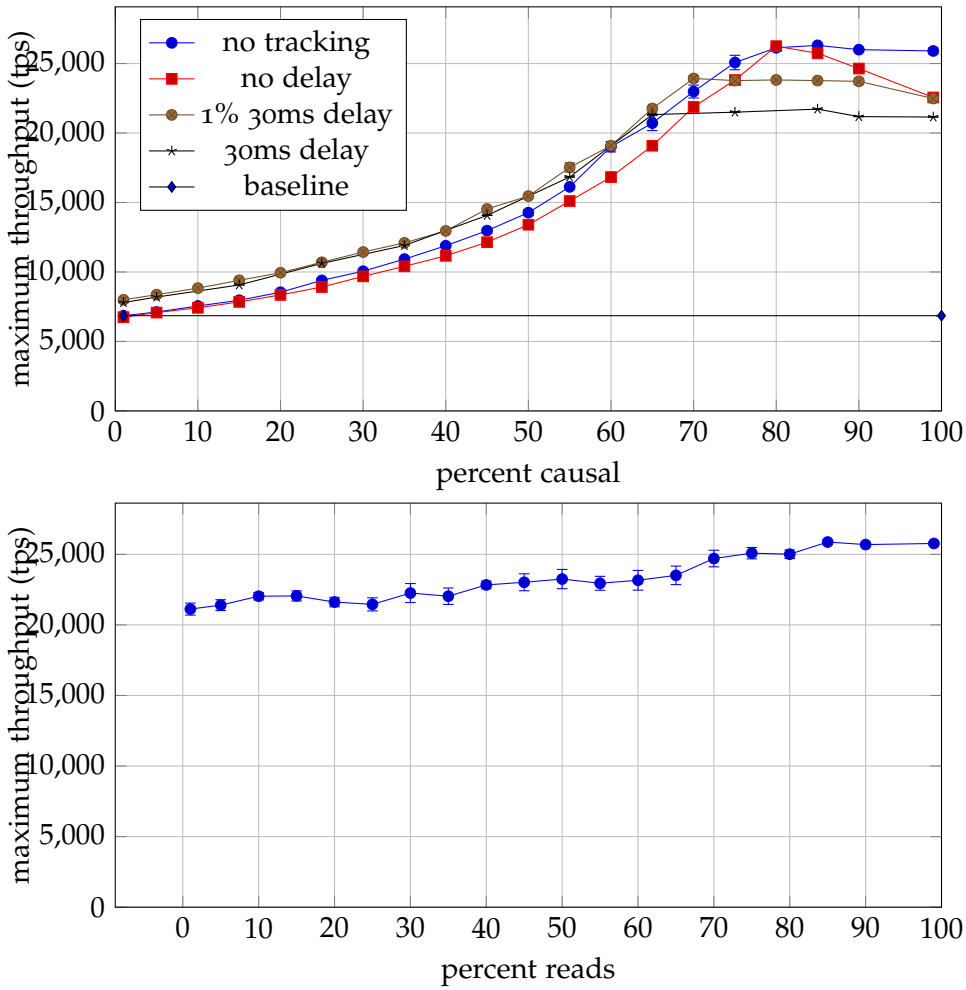


Figure 1.18: Maximum throughput as a function of linearizable mix for a 70% read workload. The blue (top circle) series shows maximum achievable throughput in transactions per second (tps) without witnesses; the remaining series shows full witness tracking with progressive artificial latency. The solid black line marks 0% causal without tracking (also the leftmost blue point), which serves as a baseline. (b) Maximum throughput as a function of read share for a 75% causal workload without tracking.

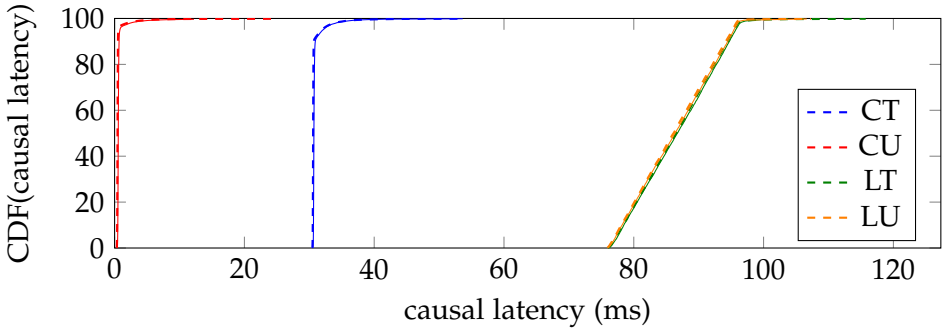


Figure 1.19: CDF plots for operation latency. C: Causal, L: Linearizable, T: Tracked, U: Untracked. Dashed lines: reads, solid lines: writes. All linearizable lines appear atop each other on the right.

a speedup compared to the simple alternative of running transactions entirely with linearizability. fig. 1.18 shows that, indeed, mixing causal and serializable operations considerably increases maximum throughput.

Because of the high latencies incurred by serializable transactions, increasing the causal percentage of overall operations yields significant performance improvements. These benefits level off at about 80% causal in our tests; at this point, the causal storage system becomes overloaded, limiting the benefits of lower latency.

OVERHEAD OF WITNESSES One concern about MixT might be the overhead introduced by witnesses. We modify our simple counter increment test to explicitly include both linearizable and causal phases in transactions which follow a consistency mode switch, and to force witness generation in these transactions even if they would not normally require it. The rationale for this is based on the non-compositionality of causal consistency. We additionally modify our experimental setup to simulate latency of replication, first forcing approximately 1% of causal witness verifications to delay for 30ms, then explicitly delaying all causal witness

verification requests by 30ms. As seen in fig. 1.18, the witness mechanism has a noticeable impact mostly above 60% causal, with a maximum slowdown of approximately 10%: well above the performance possible were the entire transaction mix to remain linearizable. Further demonstrating that round-trip time is paramount, read-only transactions achieved only 20% higher maximum throughput than read-write transactions.

LATENCY Witnesses do affect latency, especially because of design decisions made for backward compatibility. Figure 1.19 shows this effect. Latencies are presented as a CDF collected from the system running at 60% of maximum throughput, in a configuration in which 75% of all operations are reads and 75% of all operations use the causal store. To see the worst-case impact of witnesses, we ran this test twice: once with witnesses enabled and a forced 30ms delay ("tracked"), and once without ("untracked"). The red (leftmost) and orange (rightmost) lines on this graph measure performance without witnesses; the green (also rightmost) and blue (middle) lines represent performance with witnesses. The forced 30ms delay on witnesses is quite clear for causal operations, but there is almost no other overhead. On the other hand, the impact of witnesses on linearizable operations is negligible; as witnesses incur no replication delay in the linearizable store, they never delay linearizable phases.

In all configurations, a very small number of requests (less than .01%) experienced extreme latency—as high as 30s in the worst case. We believe this to be an artifact of unfairness in Postgres's contention management.

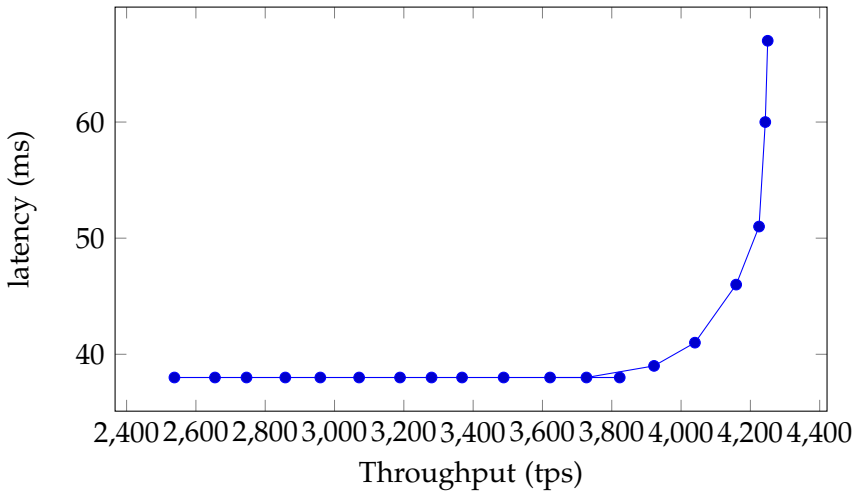


Figure 1.20: Throughput vs. latency for Message Groups.

Table 1.1: Maximum throughputs for Message Groups, with standard error. The rightmost four columns give the number of reads (R) and writes (W) and indicate whether the transaction involves a causal (C) or linearizable (L) phase.

Transaction	Throughput (tps)	R	W	C	L
Check inbox	10,626 ± 15	6	0	✓	×
Join group	5,430 ± 30	2	1	✓	✓
Deliver message	3,313 ± 4	6	3	✓	✓
Create user	972 ± 19	5	2	✓	✓

1.9.4 *Message Groups Results*

To evaluate the running Message Groups example, we use a configuration with 40,000 groups, each of which contains a single distinct user. Each of these 40,000 users has a single message in their inbox. We disable message logging, eliminating eventually-consistent phases and leaving only causal and linearizable phases. We first run each Message Groups transaction in isolation against this initial configuration, establishing an average maximum throughput over at least 3 runs (table 1.1). This table lists the average

maximum throughput for each transaction in isolation, along with the number of read and write operations executed during these transactions. For all transactions, we report the number of operations executed when in our initial configuration; message delivery and inbox downloading require more operations as the group and inbox sizes grow. The purely causal inbox download transaction benefits from the speed of causal consistency, while the mixed-consistency transactions all achieve reasonable performance despite the overhead of contacting a distant linearizable store.

We also evaluate performance on a mix of transactions: 56% inbox checking, 20% message posting, 18% group joining, and 6% user creation. We evaluate the system for 3 minutes, slowly increasing the client request rate from 2,000 tps to 5,000 tps. Average maximum throughput over 4 trials was 4,237 $\pm$ 10.5 tps with an abort rate between .0161% and .0187%. This represents a speedup of 3.5 over a baseline in which all operations execute against the linearizable store (average maximum throughput: 1,228 $\pm$ 15 tps). As expected, mixed-consistency transactions yield significant speedup.

1.10 RELATED WORK

Quelea [Sivaramakrishnan, Kaki, and Jagannathan, 2015] and Disciplined Inconsistency [Holt, Bornholt, et al., 2016] are the work closest to MixT in spirit. Both use user-provided data annotations to infer appropriate consistency levels for operations within a single program. The system then automatically chooses an appropriate consistency model for each operation. The Quelea approach, using Cassandra to store compressed

logs of all system events, differs markedly from MixT's approach of transaction partitioning based on static information flow analysis. Disciplined Inconsistency also uses information flow to enforce separation between consistency labels but does not offer any transactional mechanism. All three systems solve distinct slices of distributed, mixed-consistency programming, suggesting a combination of approaches is an avenue for future work.

CHOOSING CONSISTENCY LEVELS Choosing appropriate consistency models for data is a problem orthogonal to our work. Prior work [Gotsman et al., 2016; Herlihy, 1991; Holt, Bornholt, et al., 2016; Holt, Zhang, et al., 2015; C. Li, Leitão, et al., 2014; C. Li, Porto, et al., 2012; Sivaramakrishnan, Kaki, and Jagannathan, 2015] provide languages of constraints to describe data invariants, in turn providing the weakest consistency possible while still satisfying those constraints. Other work [Brutschy et al., 2017; Gotsman et al., 2016; Kaki et al., 2017] aims for formal methods for users to reason about their choice of consistency level and to prove that desired code invariants are satisfied.

TRANSACTIONS IN WEAK GEO-REPLICATED SYSTEMS Existing work in the shared memory [Dongol, Jagadeesan, and Riely, 2018] and distributed systems [Ardekani, Sutra, and Shapiro, 2013; Du et al., 2014; Hsu and Kshemkalyani, 2018; Lloyd et al., 2013; Mehdi et al., 2017; Shudo and Yaguchi, 2017] communities has attempted to provide single-store transactions in the presence of weak consistency guarantees. This prior work focuses on definitions and mechanisms for weak transactions at a single consistency level, and indeed, we rely on the guarantees they provide.

MIXED-CONSISTENCY SYSTEMS Many existing data stores provide operations with a variety of consistency guarantees [Cooper et al., 2008; DeCandia et al., 2007; Lakshman and Malik, 2010; C. Li, Porto, et al., 2012; Plugge, Membrey, and Hawkins, 2010], but without providing any semantic guarantees across operations. Others provide tools to tune consistency based primarily on performance considerations [Chatterjee and Golab, 2017; D. B. Terry, Prabhakaran, et al., 2013; Yang, You, and Gu, 2017]. Guerraoui, Pavlovic, and Seredinschi [2016] define a unique programming model by which programs are first presented with weakly consistent data and may choose to wait for strong data instead. These systems provide neither general transaction mechanisms nor strong semantic guarantees.

MIXED-CONSISTENCY SYSTEMS WITH TRANSACTIONS Previous work [Gao et al., 2003; Kraska, Hentschel, et al., 2009; Kraska, Pang, et al., 2013; C. Li, Porto, et al., 2012; Yang, You, and Gu, 2017] focuses on progressively weakening transaction isolation based on a combination of run-time and static analysis, with the aim of enforcing strong consistency. Several papers provide mechanisms for users to choose transaction isolation levels [Brutschy et al., 2017; Kaki et al., 2017; Xie, Su, Kapritsos, et al., 2014], but do not handle the semantic anomalies involved. A few systems [Dongol, Jagadeesan, and Riely, 2018; Yang, You, and Gu, 2017] provide distributed transactions at multiple consistency levels, but allow unsafe mixing of consistency levels. Microsoft's new database Cosmos DB is a recent example, providing transactions with a choice of four well-defined consistency levels [Gentz et al., 2017]. Some prior systems do enable programmers to mix transactions of different consistency with strong guarantees [C. Li, Porto, et al., 2012; Shasha et al., 1995; Xie, Su, Littley, et al., 2015]. However,

this line of work relies on a closed-transaction model wherein the system is aware of all possible transactions any client will run; performance is brittle because changing a single transaction somewhere in the system can significantly affect the performance of unrelated transactions. This work cannot mix consistency within a single transaction, and it focuses on a single store. Nevertheless, these systems could be used by MixT as backing stores.

CONSISTENCY ACROSS MULTIPLE SYSTEMS A much smaller body of work attempts to make the job of programming against multiple data stores easier. The most obvious candidate is SQL, and the SQL compatibility libraries like JDBC and ODBC [Hamilton, Cattell, Fisher, et al., 1997]. These standardized languages attempt to provide a unified API for programming against every RDBMS; and while each different database has its own unique implementation of the SQL standard, much of the language is shared, making it easy to port simple code which ran against one RDBMS to run against a different one. The SQL language itself is only aware of a single database system, leaving the work of coordinating actions across multiple database systems up to the programmer. Additionally, issues of consistency and isolation level are not addressed in SQL itself; rather, each underlying system determines which actions are safe.

ENHANCING CONSISTENCY OF EXISTING SYSTEMS Beyond SQL, some existing work has focused on mechanisms that upgrade the consistency guarantees of weakly consistent underlying stores [Bailis, Ghodsi, et al., 2013; Hsu and Kshemkalyani, 2018; Shudo and Yaguchi, 2017]. Indeed, several projects [Lloyd et al., 2013; Sivaramakrishnan, Kaki, and

Jagannathan, 2015] use this approach internally, adding consistency layers atop existing distributed systems like Cassandra.

1.11 CONCLUSION

We have introduced a new domain-specific programming language for writing modern geodistributed applications that need to trade off performance and consistency. The mixed-consistency transactions offered by MixT make it possible for programmers to safely combine data from multiple consistency levels in the same transaction, with confidence that weaker data does not corrupt the guarantees of stronger data. Appealingly, this model can be implemented in a backward-compatible way on top of existing stores that offer their own distinct consistency guarantees, without disrupting legacy applications on those stores. The performance results suggest that for geodistributed applications, mixed-consistency transactions enable higher performance by using weaker consistency models selectively and safely.

2

DERECHO

2.1 INTRODUCTION

While MixT shows how to safely combine weak and strong consistency, it would still be preferable to simply use strong consistency everywhere; the trouble is that existing strongly-consistent systems don't scale. There is a pervasive need for scalable, cloud-hosted services that guarantee rapid, accurate responses to huge event volumes. Yet today's high-throughput cloud infrastructure is focused almost entirely on weakly-consistent systems. As has been reliably demonstrated, working with weak consistency is hard; once a system hits millions of events per second, even the most unlikely consistency violations can lead to continuous errors, lost data, or undebuggable performance regressions. With MixT (chapter 1), programmers can annotate their data, indicating that vulnerable data must be stored under strong consistency to avoid these issues. But mixed transactions provide their greatest performance improvement when most transactions avoid touching strongly-consistent data at all. Thus programmers avoid processing their data as it streams into the system, instead storing it in some distributed filesystem for later processing. This comes at great expense, both in terms of resources devoted to this storage and in terms of latency induced by the need to wait long periods for data to converge to a reliable, strongly-consistent state.

Derecho is a fast, strongly-consistent distributed systems toolkit suitable for streaming pipelines currently relegated to the realm of weak consistency. Derecho allows programmers to describe their distributed system as a constellation of replicated objects—effectively actors ([Agha, 1986; Hewitt, Bishop, and Steiger, 1973])—which can call out to each other via a custom remote method invocation (RMI) framework. To achieve peak performance, Derecho allows programmers to directly control replication factor, sharding, and layout on a per-object basis, allowing programmers to transparently group objects which would benefit from co-location or ensure independent services live on independent machines.

Once a Derecho constellation (or, more prosaically, "group") has been established, Derecho gets out of the way and lets the replication fly. Derecho's namesake is an intense storm characterized by powerful straight-line winds that overwhelm any obstacle. This work views data replication similarly: Object replication and communication occurs over non-stop pipelines, without pausing for consensus or coordination in the absence of failures. To achieve strong consistency Derecho overlays an out-of-band state-machine model [Schneider, 1990] on its replicated objects, reporting data to listening applications only when it has been durably and consistently replicated. Derecho's control messages, handling consistency and fault tolerance, are implemented as a separate subsystem from its replication protocol, allowing both to proceed at their maximum rate.

Key to both Derecho's correctness and performance are its replication and membership management protocols. We adapted the tried-and-true protocols from virtual synchrony [K. P. Birman and T. A. Joseph, 1987] to a new platform: a *monotonic* Datalog-like core language implemented atop a lock-free distributed Shared State Table (SST) [Gallaire and Minker,

1978]. By leveraging monotonicity, Derecho's replication protocols never need to block on a consensus leader for correctness; the system only requires a leader to coordinate membership changes and system restart. We achieve this by phrasing stable message delivery—the key primitive of virtual synchrony—in terms of stable predicates implemented in the SST's monotonic core language. Thus data is moved in non-blocking, high-rate flows; control information is exchanged through non-blocking one-way flows; and the update and query paths are separated so that neither blocks the other.

Our experiments show that Derecho is orders of magnitude faster than today's most widely-used Paxos libraries and systems, even when running over TCP—a domain for which Derecho is not optimized. On a machine where the C++ `memcpy` primitive runs at 3.75GB/s for non-cached data objects, Derecho over 100Gbps RDMA can make 2 replicas at 16GB/s and 16 replicas at 10GB/s: far faster than making even a single local copy. The slowdown is sublinear as a function of scale: with 128 replicas, Derecho is still running at more than 5GB/s. Latencies from when an update is requested to when it completes can be as low as 1.5μs, and recipients receive them simultaneously, minimizing skew for parallel tasks.

This new point in the cloud-performance space enables new options for cloud infrastructure design. By offering consistency at high speed, Derecho eliminates the need for delayed event processing, enabling distributed concurrent data-sharing patterns previously relegated to the space of NUMA architectures. And by building out this performance with a flexible API focused around replicated objects, Derecho brings together the ease-of-use of traditional actor programming with the speed of modern streaming databases.

The remainder of this chapter is organized as follows. Section 2.2 focuses on the application model, the Derecho API and the corresponding functionality. Section 2.3 discuses the SST and its core monotonic language. Section 2.4 discusses the system protocols and implementation details. Section 2.6 focuses on Derecho's performance, but also includes side-by-side comparisons with LibPaxos, Zookeeper and APUS. Section 2.7 reviews prior work.

Derecho is a large collaboration across several groups at Cornell; this chapter seeks only to present the work performed by me (Matthew Milano), using the remainder of the Derecho system as context for that work. Much of the text in this chapter was previously presented in the Derecho Journal Paper [Jha et al., 2019]. At a high level, my contributions to the Derecho project were:

- The introduction of monotonic reasoning.
- a core DSL for implementing system protocols.
- With Ken Birman, adapting the traditional vsync protocol to Derecho.
- The Derecho programming model, including the design and initial implementation of its replicated object layer.
- The Derecho verification effort in Ivy, though a majority of this work was performed by Ken Birman and would not have been possible without the assistance of Sagar Jha, Orr Tamir, and Oded Padon.

In particular, I must acknowledge Sagar Jha, Derecho's lead implementer and performance architect, who has taken decisive charge of the Derecho system's more recent extensions.

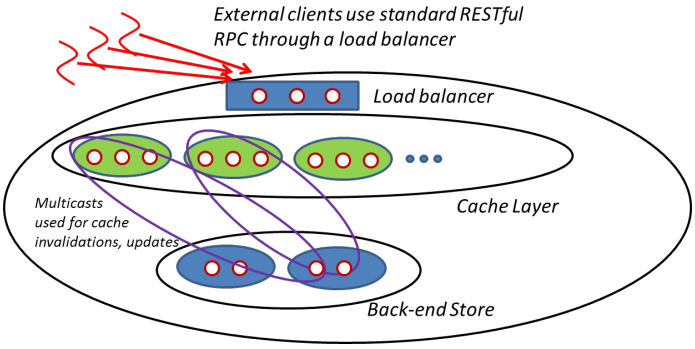


Figure 2.1: Derecho applications are structured into subsystems. Here we see 16 processes organized as 4 subsystems, 3 of which are sharded: the cache, its notification layer, and the back-end. A process can send 1-to-1 requests to any other process. State machine replication (atomic multicast) is used to update data held within shards or subgroups. Persisted data can also be accessed via the file system, hence a Derecho service can easily be integrated into existing cloud computing infrastructures.

2.2 APPLICATION MODEL AND ASSUMPTIONS MADE

2.2.1 *Target Domain*

Derecho is a library designed for single-datacenter distributed system deployments. Derecho's API allows programmers to build groups of replicated objects which represent individual services. Developers can use Derecho's object-oriented architecture to build next-generation versions of ZooKeeper (configuration management), Kafka (message queues), Ceph (filesystems), or any other service whose core primitives require replication and consistency.

But Derecho can do more than serve as a backdrop to existing microservice architectures. These existing applications evolved without consistent, near-line-speed replication; simply porting the underlying infrastructure to run over Derecho would make limited use of the system's power.

We conjecture that the most exciting possibilities will involve time-critical applications arising in the frontiers of computing. For example, emerging cyber-physical systems frequently need to make split-second decisions using only local data and computational power; a self-driving car cannot wait for the cloud of today to tell it if it's about to hit a pedestrian. We envision a future cloud with low latencies and pervasive connectivity, allowing it to serve a role even in these settings. When that happens, Derecho will be there, ready to provide microsecond-scale responsiveness without sacrificing consistency.

2.2.2 *Programming Model*

Derecho's programming model fuses the classic idea of *process groups* with the actor model. A Derecho application consists of a single top-level *group* composed of many individual *subgroups*, each serving as an independent subsystem within the overall Derecho process. Subgroups are object-oriented; each subgroup is represented as an instance of some replicated class, and subgroups communicate by issuing RMI invocations amongst themselves. In this way one can view subgroups as replicated actors, and view an entire Derecho process as a constellation of communicating actors.

When building a Derecho program, programmers design their subgroups as traditional C++ classes with one major exception: programmers do not directly construct instances of these classes, but rather defer to the top-level Derecho group to instantiate them based on programmer-specified layout constraints. These constraints are very flexible; while

programmers may specify the exact sets of nodes for each subgroup if necessary, other members of the Derecho team have provided a library of common high-level constraints to make specifying group layout flexible and quick [Jha et al., 2019]. For example, programmers could specify that a subgroup have a replication factor of at least 3, or that a certain subgroup only be provisioned if sufficient nodes in the top-level group exist to support it. To enable this flexibility the code for each subgroup must be available at all nodes which might instantiate that subgroup. To facilitate this, Derecho makes the simplifying choice of using the same binary for all nodes within the system. At startup time, each Derecho node will use the layout function to choose the roles it inhabits.

When specifying subgroup layouts, programmers also may specify unique identifiers, indexed by type, to refer to individual subgroups. Subgroups use the top-level group as a naming service capable of translating these identifiers into temporary *subgroup handles*. When viewing subgroups as actors these subgroup handles correspond to external actor references, and can be used to invoke methods on the remote object. Programmers can choose to remotely invoke methods on a single replica via *P2P* invocations, or on all replicas via *Ordered* invocations. Ordered invocations are *atomic* and *isolated*: all members of each subgroup will receive the invocation simultaneously, and no other methods may be invoked on that subgroup until the invocation has completed. The abstraction of atomicity is preserved even in the presence of failures: if a membership reconfiguration is triggered during the invocation of a remote method, then that invocation will be restarted once the reconfiguration is complete.

The choice to treat the top-level group as a naming service also grants flexibility to applications. As the system configuration changes, the ex-

act mapping from nodes to subgroups will change to better reflect the available system resources. By design, these changes invalidate all existing subgroup handles, causing their use to throw an exception. This in turn communicates to the programmer that the exact subgroup with which they are communicating has changed, allowing the programmer to either request a new subgroup handle from the top-level group, or change its behavior in light of the new system state. It is important to note that subgroup handles may only be invalidated in-between method invocations; a handle retrieved and used during a single method invocation will never be invalid.

Figure 2.1 contains an example application built using Derecho. In this example, the μ-services include a load balancer, a sharded cache layer, a sharded back-end data store, and a pattern of shard-like subgroups used by the back end to perform cache invalidations or updates. Each μ-service is implemented by an elastic pool of application instances. Notice that some μ-services consist of just a single (perhaps large) subgroup while others are sharded, and that shards typically contain 2 or 3 members. Figures 2.3-2.6 illustrate a few of the communication patterns that a subgroup or shard might employ.

2.2.3 *Restarts and Failures*

Derecho can tolerate two forms of failure: *benign crash failures* and *network partitioning* (see [Jha et al., 2019] more details). We believe that such failures will occur relatively infrequently for even moderately large data center services (ones with thousands of processes). At Google, Jeff Dean measured

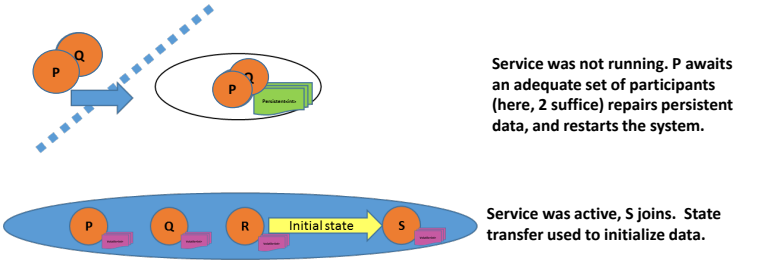


Figure 2.2: When launched, a process linked to the Derecho library configures its Derecho instance and then starts the system. On the top, processes P and Q start a service from scratch; below, process S joins a service that was already running with members {P,Q,R}. Under the surface, the membership management protocol uses leader-based consensus, with the lowest-ranked node (here P) as the initial leader.

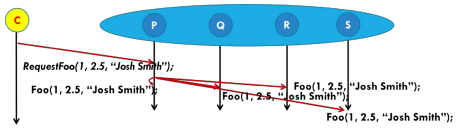


Figure 2.3: Multicasts occur within subgroups or shards and can only be initiated by members. External clients interact with members via P2P invocations.

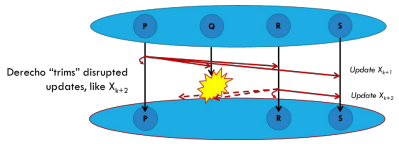


Figure 2.4: If a failure occurs, cleanup occurs before the new view is installed. Derecho supports several delivery modes; each has its own cleanup policy.

reliability for several such services, and found that disruptive failures or reconfigurations occurred once per 8 hours on average [Shankland, 2008]. Derecho needs just a few hundred milliseconds to recover from failure.

The transports over which Derecho runs all deal with packet loss[1]; hence Derecho itself never retransmits. Instead, unrecoverable packet loss results in a failure detection even if both endpoints are healthy. The communication layer notifies Derecho, which initiates reconfiguration to drop the failed endpoint.

1 RDMA requires a nearly perfect network, but does have a very primitive packet retransmission capability.

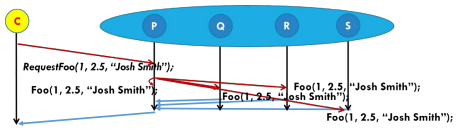


Figure 2.5: A multicast initiated within a single subgroup or shard can return results.

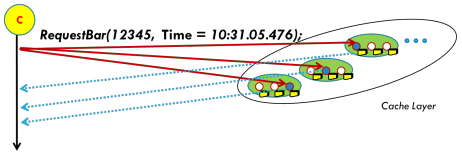


Figure 2.6: Read-only queries can also occur via P2P invocations that access multiple subgroups or shards.

A network failure that partitions a group would manifest as a flurry of "process failure" events reported on both "sides" of the failed link. As is standard in this space, Derecho's virtual synchrony protocol will cause all non-majority partitions to shut down, leaving only at most a single majority partition to continue. This majority partition will re-configure itself, dropping all unreachable group members. In terms of Brewer's CAP conjecture, this places Derecho firmly into the "CP" bucket [Brewer, 2010; Gilbert and N. Lynch, 2002]. We will review both failure handling and the membership reconfiguration in more detail in section 2.4.1.

Even if the top-level group never partitions, all the members of a shard or subgroup could fail. In such a setting, the top-level group would be unable to reprovision the subgroup; indeed, Derecho pauses even if a single shard has fewer than its minimum number of members. We do this to respect programmer-specified constraints; allowing updates to an under-provisioned subgroup might violate the developer's desired degree of replication, or fail to adhere to machine placement constraints.

Accordingly, Derecho waits for an *adequate* next view after failure: one that includes a majority of members of the prior view, and in which all shards have a sufficient number of members. If needed, Derecho then copies state to joining members, and only then does the system permit

normal activity to resume. This late-joining protocol, and its associated restart protocols, are due to Tremel and can be found in more detail in [Jha et al., 2019].

2.2.4 *The `replicated<T>` Class*

As mentioned in subsection 2.2.2, each Derecho subgroup can be viewed as an actor, backed by some existing C++ class. These actors are instances of a `replicated<T>`, a wrapper around some user-provided class. These actors are managed by Derecho; users access instances of these `replicated<T>`s by requesting *handle*s for individual subgroups from the top-level group. Any class is in principle suitable to replication as a `replicated<T>`; the only requirements enforced by Derecho are that the class is serializable and that the programmer explicitly annotates methods which should be exposed via RPC. Derecho provides macros which can automatically generate both the requisite serialization code and the necessary method annotations.

For example, if some processes wanted to invoke a method on subgroup `k` of type `MemCacheD`, they would first look up a handle for that subgroup from the top-level group:

```
ExternalCaller<MemCacheD>& cache = g.get_nonmember_subgroup<MemCacheD>(k);
```

In this example, the caller itself is not necessarily a member of the target group, and so must request an `ExternalCaller` handle for this group.

To use this handle, the caller might choose to issue a RMI to a specific replica within the group using the lightweight `p2p` communication protocol. Here, the caller communicates specifically with a replica named `who`:

```
auto outcome = cache.p2p_send<RPC_NAME(request_put)>(who, "John Smith",
    22.7);

auto result = cache.p2p_send<RPC_NAME(get)>(who, "Holly Hunter");
```

The `ExternalCaller` handle is limited to these single-replica invocations. Invoking an ordered (i.e., *atomic*) operation across all members of a group requires a first-class `replicated<T>` reference:

```
replicated<MemCacheD>& cache = g.get_subDerechoGroup<MemCacheD>(k);
auto outcome = cache.ordered_send<RPC_NAME(put)>("John Smith", 22.7);
```

Here the caller used `ordered_send`, which provides a 1-to-N atomic multicast. Failures that disrupt an `ordered_send` are masked: the protocol cleans up, then reissues requests as necessary in the next adequate membership view, preserving sender ordering.

In Derecho, Remote Method invocations follow a promise/future pattern; the `outcome` object contains a future, which can be queried to determine the status of the request and used to retrieve its result when the request is complete. Derecho exposes each replica's response to the multicast individually; if desired, programmers can use the `outcome` object to iterate over each response as it arrives.

2.3 A CORE MONOTONIC LANGUAGE

The overarching trend that drives our system design reflects a shifting balance that many recent researchers have highlighted: RDMA networking

is so fast that to utilize its full potential, developers must separate data from control, and cannot wait for traditional consensus operations in the critical path [Belay et al., 2014; Peter et al., 2014]. This is the same argument at the heart of the popularity of weak consistency: the system must manage replicated data, but cannot wait for these replicas to fully synchronize after every operation. Our solution to this seeming contradiction comes in the form of a new core *monotonic* programming language, which guarantees convergence in the style of Bloom^L [Alvaro, Bailis, et al., 2013]. This language allows programmers to write stable predicates over a series of *monotonic variables* that express *monotonic deductions*, and to condition visible actions (or *triggers*) on those predicates.

2.3.1 *Monotonic Deduction on Asynchronous Information Flows*

Monotonic deductions are a natural match for protocols which exchange *stable* knowledge about system state—for example whether a specific message has reached at least N participants, or whether a control message has received acknowledgment from $f + 1$ replicas. These properties are examples of *stable predicates*; once they become true, they will remain true for all time. Control decisions which are based *only* on stable predicates will therefore themselves be stable; no matter how far behind a replica is, if it takes an action based only on some true stable predicates, then that action cannot have depended on any messages the replica has yet to receive, and thus was safe to take.

For example, we could rephrase Paxos as a sequence of stable deductions over monotonic system state. We begin by phrasing control information

as set of counters and booleans which track which messages have been delivered, received, or acknowledged by each replica, and which failures are currently suspected by each replica. We then use simple *monotonic functions*—like max, min, and addition—to aggregate over shared counters and booleans. Critically, monotonicity is compositional; as these operations preserve the ordering of their inputs, all predicates comprised of compositions of these operators will always, constructively, be stable. Under the reasonable assumption that the boolean values `false` and `true` are ordered with `false` $<$ `true`, if some monotonic computation returns `true` in some state, then for it to return `false` in some greater state would require the computation to produce a *lesser* value on *greater* input; this contradicts monotonicity, and would only be possible if one of the constituent computations was non-monotonic. And as the underlying data is itself monotonic, we can be assured that, at all points, all later states are in fact greater than (or equal to) the current state.

Monotonic predicates permit discovery of *ordered deliverability* or *safety* for sets of messages, which can then be delivered as a batch. Notice that this batching occurs on the receivers, and will not be synchronized across the set: different receivers might discover safety for different batches. The safety deductions are all valid, but the batch sizes are accidents of the actual thread scheduling on the different receivers.

In contrast, many Paxos protocols use batching, but these batches are formed by a leader. A given leader first accumulates a batch of requests, then interacts with the acceptors (log managers) to place the batch into the Paxos log. Batches are processed one by one, and incoming requests must wait at various stages of the pipeline.

Monotonic protocols can achieve high efficiency. Yet notice that the core question of safety is unchanged: our monotonic phrasing of consensus still uses a standard Paxos definition. In effect, we have modified the implementation, but not the logical guarantee.

2.3.2 *Introducing the Shared State Table*

To support this monotonic reasoning, at the core of Derecho lies a dedicated distributed abstraction: a *shared-state table*, or SST. The SST offers a tabular distributed shared memory abstraction. Every member of the top-level group holds its own replica of the entire table, in local memory. Within this table, there is one identically formatted row per member. A member has full read/write access to its own row, but is limited to read-only copies of the rows associated with other members. This simple model provides shared memory while eliminating write-write contention on memory cells, as any given SST row only has a single writer.

Even though any given SST cell has just one writer, notice that a sequence of updates to a single SST cell will overwrite one another. If writes occur continuously, and the reader continuously polls its read-only copy of that cell, there is no guarantee that they will run in a synchronized manner. Thus a reader might see the values jump forward, skipping some intermediary values. To handle this, we build a language atop the SST which ensures all values in the SST are updated and used *monotonically*, preventing computations over the SST from being sensitive to these skips.

2.3.3 *A Monotonic Language for the SST*

Built atop the SST is a core language of monotonic combinators, implemented as a library DSL entirely within C++.

2.3.3.1 *Projectors and Reducers*

The simplest construct in this core language is the *projector*, a wrapper which allows programmers to project a value from a row. Generally, projectors just access some field within the row, although they may perform more complex reasoning (for example, indexing into a vector). These projectors are functions with the type $\mathsf{Row} \to \mathsf{T}$; their primary purpose is to lift cell access into the combinator language, though they also offer a convenient place to support variable-length fields and to implement any needed memory barriers. To ensure the correctness of the program, all projectors must be monotonic. It is an error to implement a non-monotonic projector, though as projectors may need to access low-level systems code we cannot statically enforce monotonicity.

The second SST tool is the *reducer* function, SST's primary mechanism for resolving shared state. A reducer function's purpose is to produce a summary of a certain projector's view of the entire SST; intuitively, it is run over an entire SST column. Reducers have the signature $\texttt{column} \to \texttt{T}$, where a `column` is produced by mapping a projector over a `vector[Row]`. One can think of reducers as serving a similar role to "merge" functions often found in eventual consistency literature; they take a set of divergent views of the state of some datum and produce a single summary of those views. One could also think of reducers as akin to folding an

operator over a projected column. As with projectors, reducers must be monotonic; Aggregates such as min, max, and sum are all examples of reducer functions.

One can lift a Projector into a Reducer by running the projector on a single row, and ignoring the remaining rows. Conversely, one can build a Projector out of a Reducer by creating a new column in which to store the Reducer's result, and returning the projector which accesses this new column.

2.3.3.2 *Composing programs*

New Projectors and Reducers can be built by chaining existing projectors and reducers. Chaining projectors is straightforward; as each projector has type $T \to U$ for some T and U, one can directly compose a $T \to U$ projector with some other $U \to V$ projector through standard function composition. Applying a projector to the result of a reducer is similarly straightforward.

Much more interesting is the ability to compose reducers themselves. In the SST's core language, composition of reducers corresponds to distributed computation. To compose two reducers r_1 and r_2, one first takes r_1 and run it at every replica, collecting the results into a vector. This vector becomes a new "virtual" column in the SST table, upon which r_2 may now run. This "distributed execution" composition effectively takes a reducer function of type $\texttt{column}[T] \to U$, and maps it over all the replicas, resulting in a new function of type $\texttt{vector}[\texttt{column}[T]] \to \texttt{column}[U]$. As the output of this lifted reducer is itself a column, it matches the type required for the input of a reducer, allowing a reducer to be sequenced after it. Our

language automatically introduces this lifting to handle composition of reducers.

Note that this is quite similar to the process for converting reducers to projectors outlined previously; the key difference is that the naive lifting leaves the remainder of the new column blank, while the distributed execution fills the entire column with the symmetric execution of the reducer at all replicas. Both options are available; a programmer may choose to explicitly chain reducers without a distributed execution via an explicit naive lifting.

As monotonicity is compositional, any composition of monotonic projectors and reducers will itself be monotonic. Thus by combining reducers and projectors, Derecho's protocols can employ complex predicates over the state of the entire SST without reasoning directly about the underlying consistency. Let's look at an example.

```
struct SimpleRow {int i;};
int iget(const volatile SimpleRow& s){
  return s.i;
}
Projector<int> proj(){
  return (Min(as_projector(iget)) > 7 ) || (Max(as_projector(iget)) < 2);
}
```

Here, function `proj` lifts the function `iget` into a projector, calls the reducers `Min` and `Max` on this projector, then uses the boolean operator reducers to further refine the result. This defines a new composite projector, returned by proj. This projector reads from a "virtual" column created via the distributed execution of Min and Max.

This new projector, `proj`, can do far more. For example, `proj` can be associated with a physical column in which its output will be stored. One

can also register a *trigger* to fire when the projector has attained a specific value. An extended example demonstrates this:

```
enum class Names {Simple};
SST<T> build_sst(){
  auto predicate = associate_name(Names::Simple, proj());
  SST<T> sst = make_SST<T>(predicate);
  std::function<void (volatile SST<T>&)> act = [](...){...};
  sst->registerTrigger(Names::Simple, act);
  return sst;
}
```

Here `proj` has been associated with the name `Simple` chosen from an `enum class Names`, allowing us to register the trigger `act` to fire whenever `proj` becomes true. As shown here, a trigger is simply a function of type `volatile SST<T>&` $\rightarrow$ `void` with one important restriction: it must ensure stability of registered predicates, and cannot depend on exact values in the SST without first ensuring those values are up-to-date. If the result of a trigger can never cause a previously-true predicate to turn false, reasoning about the correctness of one's SST program becomes easy. Using this combination of projectors, predicates, and triggers, one can effectively program against the SST at a nearly declarative high level, proving an excellent fit for protocols matching the common pattern "when everyone has seen event X, start the next round."

Interestingly, programming via composition of projectors and reducers has a natural comonadic structure over the SST, when the SST is viewed as a vector of Rows. We choose to demonstrate this comonad structure by defining an extract (counit), doubling, and map; composing doubling and map yields extend (cobind).

Extract is built into projectors. Projectors are always evaluated on the row "owned" by the current replica; the function which selects this row has type `vec[Row] → Row`, and satisfies the requirements of extract.

Duplication corresponds to "perfect replication": if a replica knows that its SST is identical to all other SSTs, then that replica can simulate any distributed computation by simply running it on copies of its local SST. The function which copies the SST is doubling, and it has the signature `vec[Row] → vec[vec[Row]]`.

Map corresponds to the distributed execution of reducers. Distributed execution of reducers has type `(vec[T] → U) → vec[vec[T]] → vec[U]`, producing a system-wide summary of a single column. We can generalize reducers to operate against entire rows rather than single columns, giving the type `(vec[Row] → U) → vec[vec[Row]] → vec[U]` to their distributed executor. Composing doubling and map yields cobind: `(vec[Row] → U) → vec[Row] → vec[U])`.

We believe that further exploration of this comonadic structure is warranted, and could form an excellent basis by which to prove properties of the core SST language.

2.3.4 *Considerations for Programming against the SST*

2.3.4.1 *Encoding Knowledge Protocols in SST*

SST predicates have a natural match to the *logic of knowledge* [Halpern and Moses, 1990], in which we design systems to exchange knowledge in a way that steadily increases the joint "knowledge state." Suppose that rather than sharing raw data via the SST, processes share the result of

computing some predicate. In the usual knowledge formalism, we would say that if *pred* is true at process P, then P *knows* pred, denoted $K_P(pred)$. Now suppose that all members publish the output of the predicate as each learns it, using a bit in their SST rows for this purpose. By aggregating this field using a reducer function, process P can discover that *someone knows pred*, that *everyone knows pred*, and so forth. By repeating the same pattern, group members can learn $K^1(pred)$: every group member *knows* that every other member *knowspred*.

2.3.4.2 *Stable, Monotonic Predicates*

Earlier, we defined a monotonic predicate to be a stable predicate defined over a monotonic variable v such that once the predicate holds for value v, it also holds for every $v' \geq v$. Here is further evidence that one should think about these protocols as forms of knowledge protocols. Doing so gives a sharp reduction in the amount of SST space required by a protocol that runs as a sequence of rounds. With monotonic variables and predicates, process P can repeatedly replace values in its SST row with greater ones. As P's peers within the group compute they might see very different sequences of updates, yet all processes will eventually converge on the same set of stable predicates.

For example, with a counter, P might rapidly sequence through increasing values. Now, suppose that Q is looping and sees the counter at values 20, 25, 40. Meanwhile, R sees 11, then 27, 31. If the values are used in monotonic predicates and some deduction was possible when the value reached 30, both will make that deduction even though they saw distinct values and neither was actually looking at the counter precisely when 30

Figure 2.7: SST example with three members, showing some of the fields used by our algorithm. Each process has a full replica, but because push events are asynchronous, the replicas evolve asynchronously and might be seen in different orders by different processes.

was reached. If events might occur millions of times per second, this style of reasoning enables a highly pipelined protocol design.

2.3.4.3 *Fault Tolerance and Monotonicity*

Crash faults introduce a number of non-trivial issues specific to the SST in Derecho. Failure detection needs to run alongside all other protocols, atop the same core language for the SST. This in turn means failure detection can only be based on monotonic state: when a node communicates a failure suspicion it can never revoke that suspicion lest it violate monotonicity.

To accommodate this we adopt the following approach. We dedicate an entire column in the SST to failure suspicions for each node. A cell in that column corresponds to a failure suspicion reported by the cell's row. Failure detection is simply a max operation over the column. If any node suspects a failure, then it must:

1. Freeze its copy of the SST row associated with the failed group member (this breaks its RDMA connection to the failed node);

2. Update its own row to report the new suspicion (via the "Suspected" boolean fields seen in Figure 2.7);

3. Immediately replicate its row to every other process (excluding those its considers to have failed).

We additionally register a failure propagation trigger for each failure suspicion column, causing any node which observes a suspected failure to take the same steps as though it had detected the failure itself.

Derecho currently uses hardware failure detection as its source of failure suspicions, although we also support a user-callable interface for reporting failures discovered by the software. In many applications the SST itself can be used to share heartbeat information by simply having a field that reports the current clock time and pushing the row a few times per second; if such a value stops advancing, whichever process first notices the problem can treat it as a fault detection.

Thus, if a node has crashed, the SST will quickly reach a state in which every non-failed process suspects the failed one, has frozen its SST row, and has pushed its own updated row to its peers. However, the SST's implementation (section 2.5) does not use a reliable multicast primitive for replication; this can leave an SST write partially replicated causing the SST replicas to not be identical. In particular, the frozen row corresponding to a failed node could differ if some SST push operations failed midway.

Were this the entire protocol, the SST would be at risk of logical partitioning. To prevent such outcomes, we shut down any process that suspects a majority of members of the Derecho top-level group (in effect, such a process deduces that it is a member of a minority partition). Thus, although the SST is capable (in principle) of continued operation in a minority partition, Derecho does not use that capability and will only make progress so

long as no more than a minority of top-level group members are suspected of having failed.

2.3.4.4 *Stable, Fault-Tolerant Monotonic Reasoning.*

A next question to consider is the interplay of failure handling with knowledge protocols. The aggressive epidemic-style propagation of failure suspicions transforms a suspected fault into monotonic knowledge that the suspected process is being excluded from the system: P's aggressive push ensures that P will never again interact with a member of the system that does not know of the failure, while Derecho's majority rule ensures that any minority partition will promptly shut itself down.

This conflicts with the eventual guarantees we relied on for the correctness of our monotonic reasoning. We must consider an outcome where process P discovers that *pred* holds and acts on that knowledge, but then crashes. The failure might freeze the SST rows of in such a way that no surviving process can deduce that *pred* held at P, leaving uncertainty about whether or not P might have acted on *pred* prior to crashing.

To solve this, we make two observations. First, we observe that the correctness of a system is usually phrased only in terms of the behavior of its *correct* processes; as P has acted on *pred* "after" it has failed, that action may be ignored *provided it does not impact correct processes*. In cases where such an action might in fact impact correct processes, there is a different way to eliminate this uncertainty: before acting on *pred*, P can share its discovery that *pred* holds. In particular, suppose that when P discovers *pred*, it first reports this via its SST row, pushing its row to all other members *before* acting on the information. With this approach, there are two ways of learning that *pred* holds: process Q can directly deduce

that *pred* has been achieved, but it could also learn *pred* indirectly by noticing that P has done so. If P possessed knowledge no other process can deduce without information obtained from P, it thus becomes possible to learn that information either directly (as P itself did, via local deduction) or indirectly (by obtaining it from P, or from some process that obtained it from P). If information reaches a quorum, then it will survive even if, after a crash, the property itself is no longer directly discoverable! With stable predicates, indirect discovery that a predicate holds is as safe as direct evaluation. By combining this behavior with monotonic predicates, the power of this form of indirection is even greater.

Care must be taken when using this design pattern; it must be the case that replicas can discover not just that *pred* holds, but that any predicate implied by a combination of their local state with the local state that justified *pred* will also hold. If the projectors used to compute *pred* are also used to compute other predicates, then the values of these common projectors must be reflected to the SST alongside *pred*. For projectors built by composing reducers from the monotonic SST DSL this is handled automatically; for custom projectors written in C++, it must be ensured by the programmer.

Notice also that when Derecho's majority rule is combined with this fault tolerant learning approach, P either pushes its row to a majority of processes in the epoch, then can act upon the knowledge it gained from *pred*, or P does not take action and instead crashes or throws a partitioning fault exception (while trying to do the push operation). Since any two majorities of the top-level group have at least one process in common, in any continued run of the system, at least one process would know of P's

deduction that *pred* holds. This will turn out to be a powerful tool in what follows.

2.4 MEMBERSHIP MANAGEMENT WITH VIRTUALLY-SYNCHRONOUS PAXOS

We used the SST's monotonic language to implement *virtually synchronous Paxos*, which combines a membership-management protocol that was created as part of the Isis Toolkit [K. P. Birman and T. A. Joseph, 1987] with a variation of Paxos. The virtually synchronous Paxos model was originally suggested by Malkhi and Lamport at a data replication workshop in Lugano. Later, the method was implemented in a distributed-systems teaching tool called Vsync, and described formally in Chapter 22 of [K. P. Birman, 2012].

The virtual synchrony model focuses on the evolution of a process group through a series of *epochs*. An epoch starts when new membership for the group is reported (a *new view* event). The multicast protocol runs in the context of a specific epoch, sending totally ordered multicasts to the full membership and delivering messages only after the relevant safety guarantees have been achieved. These include total ordering, the guarantee that if any member delivers a message then every non-failed member will do so[2], An epoch ends when some set of members join, leave, or fail. This could occur while a multicast is underway, resulting either in the multicast being delivered (if it had reached all surviving participants before view change) or by reissuing it (preserving sender ordering) in the next epoch.

2 The original Isis formulation added an optional early-delivery feature: multicasts could be delivered early, creating an *optimistic* mode of execution. A stability barrier (`flush`), could then be invoked when needed. Derecho omits this option.

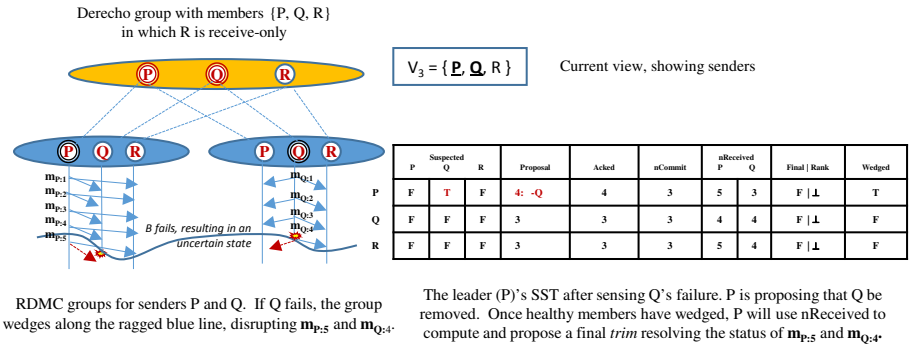


Figure 2.8: Each Derecho group has one RDMC subgroup per sender (in this example, members P and Q) and an associated SST. In normal operation, the SST is used to detect multi-ordering. During membership changes, the SST is used to select a leader. It then uses the SST to decide which of the disrupted RDMC messages should be delivered and in what order; if the leader fails, the procedure repeats.

2.4.1 *Delivery and Reconfiguration Protocols*

To avoid burdening the reader with excessive detail, we limit ourselves to a brief overview. Derecho's protocols, implemented by Birman and Jha atop the SST, can be found in full detail in the Derecho journal paper [Jha et al., 2019].

Derecho's core structure can be seen in Figure 2.8. We map the top-level group to a set of subgroups, which may additionally be sharded. Here there is see one subgroup. For each active sender, Derecho needs an SMC or RDMC session (section 2.5) that will be used to stream multicasts reliably and in sender-order to the full group; in the figure, two such sessions are in use, one from sender P and one from sender Q. The group view is seen on the top right, and below it, the current SST for the group.

The columns in the SST are used by group members to share status. From the left, there is a vector of booleans denoting failure suspicions (in

the example shown, Q has just failed, and P is aware of the event and hence "suspects" Q, shown in red). Eventually this will trigger a new view in which Q will have been removed.

Next come view-related SST columns, used by the top-level group leader to run a Paxos-based protocol that installs new views. We'll discuss this in moment.

To the right is a set of columns labeled "nReceived." These are counts of how many messages each group member has received from each sender. For example, in the state shown, R has received 5 multicasts from P, via RDMC. To minimize unnecessary delay, Derecho uses a simple round-robin delivery order: each active sender can provide one multicast per delivery cycle, and the messages are delivered in round-robin order. Derecho has built-in mechanisms to automatically send a null message on behalf of a slow sender, and will reconfigure to remove a process from the sender set if it remains sluggish for an extended period of time. Thus in the state shown, P and Q are both active, and messages are being delivered in order: P:1, Q:1, P:2, Q:2, etc.

Derecho delivers atomic multicasts when (1) all prior messages have been delivered, and (2) all receivers have reported receipt of a copy, which is determined as an aggregate over nReceived. Notice the monotonic character of this delivery rule: an example of receiver-side monotonic reasoning.

In this example, messages can be delivered up to P:4, but then an issue arises. First, P is only acknowledging receipt of Q's messages through Q:3. Thus messages up to P:4 can be delivered, but subsequent ones are in a temporarily unstable state. Further, the failure is causing the group to *wedge*, meaning that P has noticed the failure and ceased to send or deliver

new messages (wedged bit is true on the far right). Soon, R will do so as well. Q's row is ignored in this situation, since Q is suspected of having crashed.

Once the group is fully wedged by non-faulty members, the *view-change* protocol takes over. This protocol mixes aspects of monotonic reasoning (derived from the SST) with the traditional virtually-synchronous consensus protocol of [K. P. Birman and T. A. Joseph, 1987]. The lowest-ranked unsuspected process (P in this view) will propose a new view, but will also propose a final delivery "count" for messages, called a "ragged trim." P itself could fail while doing so, hence a new leader first waits until the old leader is suspected by every non-faulty group participant. Then it scans the SST. Within the SST, we include columns with which participants echo a proposed ragged trim, indicate the rank of the process that proposed it, and indicate whether they believe the ragged trim to have committed (become *final*). The sequence ensures that no race can occur: the scan of the SST will only see rows that have been fully updated.

This pattern results in a form of iteration: each successive leader will attempt to compute and finalize a ragged trim, iterating either if some new member failure is sensed, or the leader itself fails. This continues until either a majority is lost (in which case the minority partition shuts down), or eventually, some leader is not suspected by any correct member, and is able to propose a ragged trim, and a new view, that a majority of the prior members acknowledge. The protocol then commits. We have formalized this protocol as an I/O automata in the Ivy theorem prover [Padon et al., 2016].The ragged trim is used to finalize multicasts that were running in the prior view, and Derecho can move to the next view. The property just described is closely related to the weakest condition for progress in the

Chandra / Toueg consensus protocol, and indeed the Derecho atomicity mechanism is very close to the atomicity mechanism used in that protocol [T. Chandra and Toueg, 1996].

An important detail of the view-change protocol is that it is *explicitly* non-monotonic. The view-change protocol allows the leader to compute an entirely new SST, with initial values set by through the ragged trim protocol. This new SST is discontinuous with the old; it is *not* ensured that all possible stable predicates over the old SST will hold in the new. Instead, we must manually ensure that all relevant predicates remain true in the new SST, while allowing some predicates—for example failure detection—to reset. We should again emphasize that this view-change protocol is simply a rephrasing of protocols which have been proved correct in prior work. Nevertheless, our encoding of the Derecho view-change protocol in the Ivy theorem prover gives us faith that the protocol remains correct even with the changes we made to leverage monotonicity.

2.5 BACKGROUND: THE SYSTEM DETAILS OF THE SST

As a collaboration between many individuals, Derecho's core protocols are built upon abstractions which should not be included in the contributions of this thesis.

In particular Derecho moves data using a pair of zero-copy reliable multicast abstractions, SMC (due to Jha) and RDMC (due to Behrens), both of which guarantee that messages will be delivered in sender order without corruption, gaps or duplication, but lack atomicity for messages underway when some member crashes. The protocols in section 2.4.1 sense

such situations and clean up after a failure. SMC and RDMC are both single-sender, multiple receiver protocols.

These reliable multicast primitives—in particular RDMC [Behrens et al., 2018]— provide a powerful abstraction, allowing data to move across the system at unprecedented speeds. But it is indeed primitive; using RDMC requires that the application track membership and arrange for message delivery endpoints to simultaneously set up each needed session, select a sender, and coordinate to send and receive data on it. With multiple senders to the same group of receivers, RDMC provides no ordering on concurrent messages. When a failure occurs, a receiver reports the problem, stops accepting new RDMC messages, and "wedges," but takes no action to clean up disrupted multicasts.

This is where protocols implemented over the SST come in. Section 2.4.1 demonstrated how the SST's core monotonic language can be used to implement Derecho's delivery protocols. This section reviews the system details of the SST itself, and how Derecho leverages RDMA and cache-line atomicity to ensure correct and performant behavior.

Recall that the SST is a set of single-writer, multi-reader registers arranged in a table with a single writer per table row. To share data using the SST, a process updates its local copy of its own row, then *pushes* the row to other group members by enqueuing a set of asynchronous one-sided RDMA write requests. The SST also supports pushing just a portion of the row, or pushing to just a subset of other processes.

Even though any given SST cell has just one writer, notice that a sequence of updates to a single SST cell will overwrite one another. If writes occur continuously, and the reader continuously polls its read-only copy of that cell, there is no guarantee that they will run in a synchronized

manner. Thus a reader might see the values jump forward, skipping some intermediary values. This is where the monotonicity of values stored in the SST comes in: by ensuring all values in the SST are updated and read monotonically, computations over the SST cannot be sensitive to these skips.

We must still ensure, however, that this non-determinism cannot cause write-skew when reading individual values. The SST guarantees that writes are atomic at the granularity of *cache lines*, typically 64 bytes in size. The C++ 14 compiler aligns variables so that no native data type spans a cache line boundary, but this means that if an entire vector is updated, the actual remote updates will occur in cache-line sized atomic chunks. Accordingly, when updating multiple entries in a vector, we take advantage of a different property: RDMA writes respect the sender's FIFO ordering, in that multiple RDMA datagrams are applied at the target node sequentially. Thus, we can *guard* the vector within the SST with a counter, provided that we update the vector first and then the counter in a separate datagram. When a reader sees the guard change, it is safe for it to read the guarded vector elements. It can then acknowledge the data, if needed, via an update to its own SST row.

In the most general case, an SST push transfers a full row to $N-1$ other members. Thus, if all members of a top-level group were actively updating and pushing entire rows in a tree-structured network topology, the SST would impose an N^2 load on the root-level RDMA switches. Derecho takes a number of steps to ensure that this situation will not be common. Most of the protocols update just a few columns, so that only the modified bytes need to be pushed. RDMA scatter-gather is employed to do all the transfers with a single RDMA write: an efficient use of the hardware.

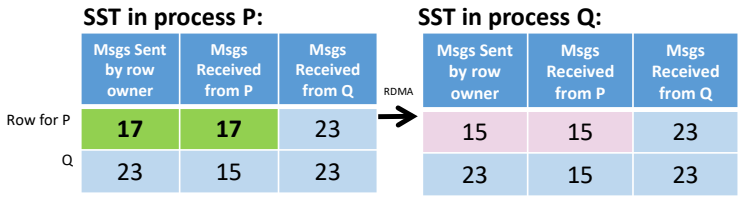


Figure 2.9: SST example with two members: P and Q. P has just updated its row, and is using a one-sided RDMA write to transfer the update to Q, which has a stale copy. The example, discussed in the text, illustrates the sharing of message counts and confirmations.

Furthermore, these updates are often of interest to just the members of some single shard or subgroup, and hence only need to be pushed to those processes. Thus the situations that genuinely involve all-to-all SST communication most often involve just 2 or 3 participants. We have never seen a situation in which the SST was a bottleneck.

2.6 PERFORMANCE EVALUATION

This performance evaluation is presented to give context to this chapter and lend credence to its claims; the only components of this analysis carried out as part of this thesis involved the overheads of Derecho's object replication layer. Other results report on the raw speed of core Derecho components [Behrens et al., 2018; Jha et al., 2019], and the speed of Derecho's replication and recovery atop those core protocols [Jha et al., 2019].

Our experiments seek to answer the following questions:

- How do the core state machine replication protocols perform on modern RDMA hardware, and on data-center TCP running over

100Gbps Ethernet? We measure a variety of metrics but for this section report primarily on bandwidth and latency.

- If an application becomes large and uses sharding heavily for scale, how will the aggregate performance scale with increasing numbers of members? Here we explore both Derecho's performance in large subgroups and its performance with large numbers of small shards (2 or 3 members, with or without overlap; for the overlap case, we created 2 subgroups over the same members). These experiments ran on a shared network with some congested TOR links, and in the overlapping shards case, the test itself generated overlapping Derecho subgroups. Thus any contention-triggered collapse would have been visible.

- Looking at the end-to-end communication pipeline, how is time spent? We look at API costs (focusing here on polymorphic method handlers that require parameter marshalling and demarshalling; Derecho also supports unmarshalled data types, but of course those have no meaningful API costs at all) and delivery delay.

- How does the performance of the system degrade if some members are slow?

- How long does the system require to reconfigure an active group?

- How does Derecho compare with other libraries for state machine replication—in particular APUS, LibPaxos and ZooKeeper?

In what follows, the small-scale experiments were performed on our local cluster, Fractus. For larger experiments, we used Stampede 1, a supercomputing cluster in Texas. Fractus consists of 16 machines running

Ubuntu 16.04 connected with a 100Gbps (12.5 GB/s) RDMA InfiniBand switch (Mellanox SB7700). The machines are equipped with Mellanox MCX456AECAT Connect X-4 VPI dual port NICs. Stampede contains 6400 Dell Zeus C8220z compute nodes with 56G (8 GB/s) FDR Mellanox NIC, housed in 160 racks (40 nodes/rack). The interconnect is an FDR InfiniBand network of Mellanox switches, with a fat tree topology of eight core-switches and over 320 leaf switches (2 per rack) with a 5/4 bandwidth oversubscription. Nodes on Stampede are batch scheduled with no control over node placement. Node setup for our experiments consists of about 4 nodes per rack. Although network speeds are typically measured in bits per second, our bandwidth graphs use units of GB/s simply because one typically thinks of replicated objects in terms of bytes.

2.6.1 *Core Protocol Performance*

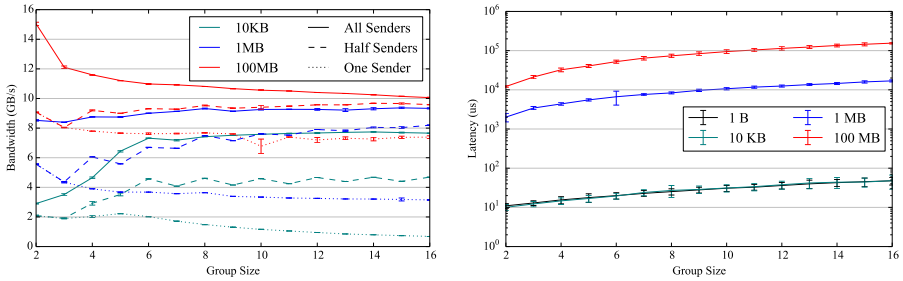
Figure 2.10 measures Derecho performance on 2 to 16 nodes on Fractus. The experiment constructs a single subgroup containing all nodes. Each of the sender nodes sends a fixed number of messages (of a given message size) and time is measured from the start of sending to the delivery of the last message. In these experiments, "senders" behave like continuous writers: all senders are constantly pumping messages into the system at the maximum achievable rate, all of which need to be totally ordered. Bandwidth is then the aggregated rate of sending of the sender nodes. We plot the throughput for totally ordered (atomic multicast) mode.

We see that Derecho performs close to network speeds for large message sizes of 1 and 100 MB, with a peak rate of 16 GB/s. In unreliable mode,

Derecho's protocol for sending small messages, SMC, ensures that we get high performance (close to 8 GB/s) for the 10 KB message size; we lose about half the peak rate in totally-ordered atomic multicast. As expected, increasing the number of senders leads to a better utilization of the network, resulting in better bandwidth. For the large message sizes, the time to send the message dominates the time to coordinate between the nodes for delivery, and thus unreliable mode and totally ordered (atomic multicast) mode achieve similar performance. For small message sizes (10 KB), those two times are comparable. Here, unreliable mode has a slight advantage because it does not perform global stability detection prior to message delivery.

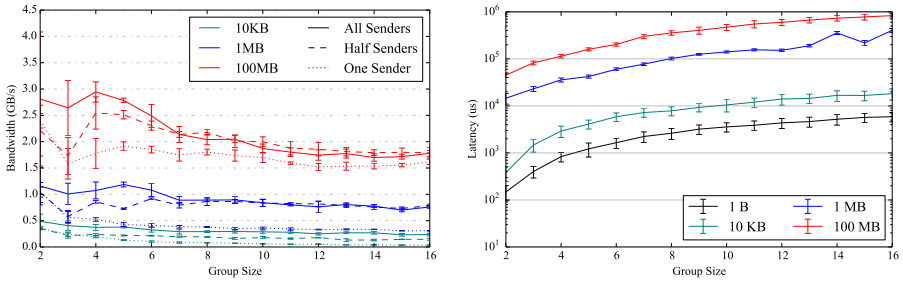
Not shown is the delivery batch size; at peak rates, multicasts are delivered in small batches, usually the same size as the number of active senders, although now and then a slightly smaller or larger batch arises. Since we use a round-robin delivery order, the delay until the *last* sender's message arrives will gate delivery, as we will show momentarily, when explaining Figure 2.15b.

Notice that when running with 2 nodes at the 100MB message size, Derecho's peak performance exceeds 12.5 GB/s. This is because the network is bidirectional, and in theory could support a data rate of 25GB/s with full concurrent loads. With our servers, the NIC cannot reach this full speed because of limited bandwidth to the host memory units.



(a) Atomic multicast throughput (RDMA) (b) Atomic multicast latency (RDMA)

Figure 2.10: Derecho's RDMA performance with 100Gbps InfiniBand.



(a) Atomic multicast throughput (TCP) (b) Atomic multicast latency (TCP)

Figure 2.11: Derecho performance using TCP with 100Gbps Ethernet.

2.6.2 *Large Subgroups, with or without Sharding*

Earlier, we noted that while most communication is expected to occur in small groups, there will surely also be some communication in larger ones. To explore this case, we ran the same experiment on up to 128 nodes on Stampede. The resulting graph, shown in Figure 2.12, shows that Derecho scales well. For example we obtain performance of about 5GB/s for 1MB-all-senders on 2 nodes, 2.5 GB/s on 32 nodes, and 2 GB/s on 128 nodes: a slowdown of less than 3x. Limitations on experiment duration and memory prevented us from carrying out the same experiment for the 100 MB case on 64 and 128 nodes. This also explains the absence of error

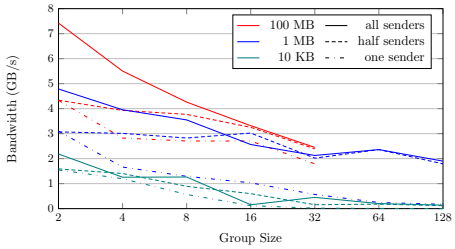


Figure 2.12: Totally ordered (atomic multicast) mode

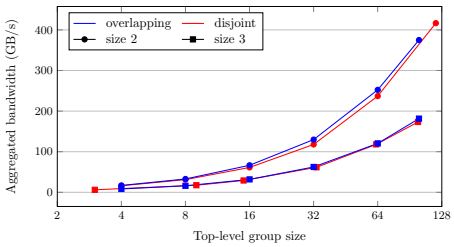


Figure 2.13: Derecho performance for sharded groups using RDMC with 100MB messages.

bars: each data point shown corresponds to a single run of the experiment. Note, however, that the performance obtained is similar to that seen on Fractus for small groups.

Next, we evaluate performance in an experiment with a large sharded group. Here, the interesting case involves multiple (typically small) subgroups sending messages simultaneously, as might arise in a sharded application or a staged computational pipeline. We formed two patterns of subgroups of fixed size: disjoint and overlapping. For a given set of n nodes, assume unique node ids from 0 to $n-1$. Disjoint subgroups partition the nodes into subgroups of the given size. Thus, disjoint subgroups of size s consist of n/s subgroups where the i^{th} subgroup is composed of nodes with ids $s*i, s*i+1, \ldots, s*(i+1)-1$. Overlapping subgroups of size s, on the other hand, place every node in multiple (s) subgroups. They consist of n subgroups where the the i^{th} subgroup is composed of nodes $i, i+1, ..., i+s-1$ (wrapping when needed).

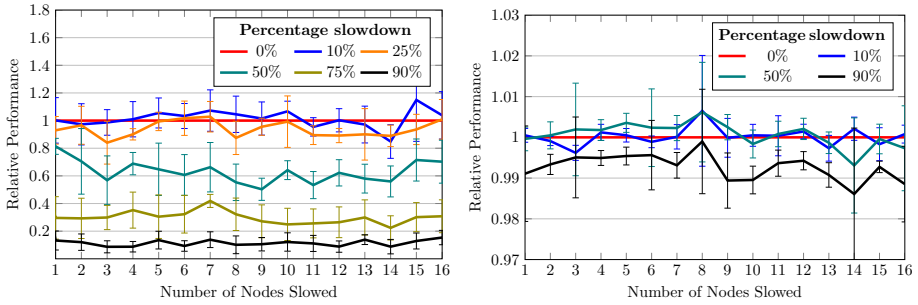
We tested with overlapping and disjoint subgroups of sizes 2 and 3. All nodes send a fixed number of messages of a given message size in each of the subgroups they belong in. The bandwidth is calculated as the sum of the sending rate of each node. Figure 2.13 shows that for large messages

(100 MB), the aggregated performance increases linearly with the number of nodes for all subgroup types and sizes. This is as expected; the subgroup size is constant, and each node has a constant rate of sending, leading to a linear increase in aggregated performance with the number of nodes.

We do not include data for small messages (10 KB) because this particular case triggered a hardware problem: the "slow receiver" issue that others have noted in the most current Mellanox hardware [Guo et al., 2016].

2.6.3 *Resilience to Contention*

The experiments shown above were all performed on lightly loaded machines. Our next investigation explores the robustness of control plane/-data plane separation and batching techniques for Derecho in a situation where there might be other activity on the same nodes. Recall that traditional Paxos protocols degrade significantly if some nodes run slowly [Marandi et al., 2014]. The issue is potentially a concern, because in



(a) 10 KB (b) 100 MB

Figure 2.14: Derecho performance for various values for efficiency and number of slow nodes, as a fraction of the no-slowdown case.

multi-tenancy settings or on busy servers, one would expect scheduling delays. In Derecho, we hope to avoid such a phenomenon.

Accordingly, we designed an experiment in which we deliberately slowed Derecho's control infrastructure. We modified the SST predicate thread by introducing artificial busy-waits after every predicate evaluation cycle. In what follows, we will say that a node is working at an efficiency of X% (or a slowdown of $(100 - X)$%), if it is executing X predicate evaluation cycles for every 100 predicate evaluation cycles in the normal no-slowdown case. The actual slowdown involved adding an extra predicate that measures the time between its successive executions and busy-waits for the adjusted period of time to achieve a desired efficiency. In contrast, we did not slow down the data plane: RDMA hardware performance would not be impacted by end-host CPU activity or scheduling. Further, our experiment sends messages without delay.

In many settings, only a subset of nodes are slow at any given time. To mimic this in our experiment, for a given efficiency of X%, we run some nodes at X% and others at full speed. This enables us to vary the number of slow nodes from 0 all the way to all the nodes, and simplifies comparison between the degraded performance and the no-slowdown case. The resulting graph is plotted in Figure 2.14.

The first thing to notice in this graph is that even with significant numbers of slow nodes, performance for the large messages is minimally impacted. This is because with large messages, the RDMA transfer times are so high that very few control operations are needed, and because the control events are widely spaced, there are many opportunities for a slowed control plane to catch up with the non-slowed nodes. Thus, for 90%

slowdown, the performance is more than 98.5% of the maximum while for 99% slowdown (not shown in the graph), it is about 90%.

For small messages (sent using SMC), the decrease in performance is more significant. Here, when even a single node lags, its delay causes all nodes to quickly reach the end of the sending window and then to wait for previous messages to be delivered. Nonetheless, due to the effectiveness of batching, the decrease in performance is less than proportional to the slowdown. For example, the performance is 70% of the maximum in case of 50% slowdown and about 15% for a 90% slowdown.

Notice also that performance does not decrease even as we increase the number of slow nodes. In effect, the slowest node determines the performance of the system. One can understand this behavior by thinking about the symmetry of the Derecho protocols, in which all nodes independently deduce global stability. Because this rule does not depend on a single leader, all nodes proceed independently towards delivering sequence of messages. Further, because Derecho's batching occurs on the receivers, not the sender, a slow node simply delivers a larger batch of messages at a time. Thus, whether we have one slow node or all slow ones, the performance impact is the same.

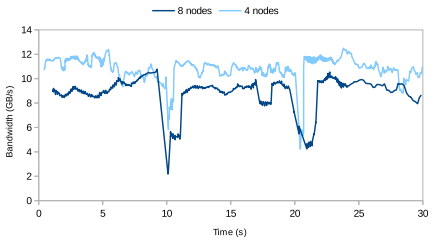
From this set of experiments, we conclude that Derecho performs well with varying numbers of shards (with just minor exceptions caused by hardware limitations), that scheduling or similar delays are handled well, and that the significant performance degradation seen when classic Paxos protocols are scaled up are avoided by Derecho's novel asynchronous structure.

Next, we considered the costs associated with membership changes.

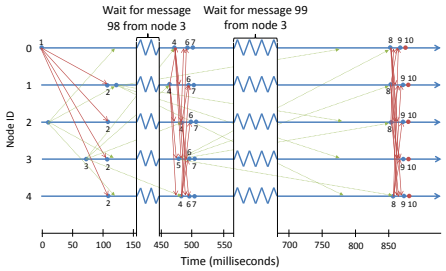
2.6.4 *Costs of Membership Reconfiguration*

In Figure 2.15a we see the bandwidth of Derecho multicasts in an active group as a join or leave occurs. The three accompanying figures break down the actual sequence of events that occurs in such cases, based on detailed logs of Derecho's execution. Figure 2.15b traces a single multicast in an active group with multiple senders. All red arrows except the first set represent some process putting information into its SST row (arrow source) that some other process reads (arrow head); the first group of red arrows, and the background green arrows, represent RDMC multicasts. At ① process 0 sends a 200MB message: message (0,100). RDMC delivers it about 100ms later at ②, however, Derecho must buffer it until it is multi-ordered. Then process 3's message (3,98) arrives (③-④) and the SST is updated (④, ⑥), which enables delivery of a batch of messages at ⑦. These happen to be messages (2,98)...(1,99). At ⑧ process 3's message (3,99) arrives, causing an SST update that allows message 100 from process 0 to finally be delivered (⑨-⑩) as part of a small batch that covers (2,99) to (1,100). Note that this happens to illustrate the small degree of delivery batching predicted earlier.

In Figure 2.16a we see a process joining: ① it requests to join, ②-⑥ are the steps whereby the leader proposes the join, members complete pending multicasts, and finally wedge. In steps ⑦-⑨ the leader computes and shares the trim; all processes trim the ragged edge at ⑨ and the leader sends the client the initial view (⑩). At ⑪ we can create the new RDMC and SST sessions, and the new view becomes active at ⑫. Figure 2.16b

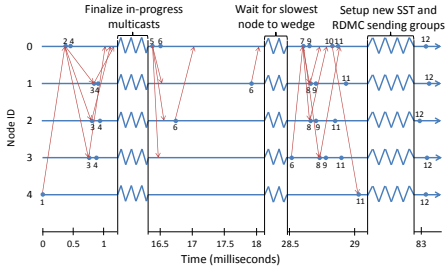


(a) Derecho multicast bandwidth with 200MB messages. A new member joins at 10s, then leaves at 20s.

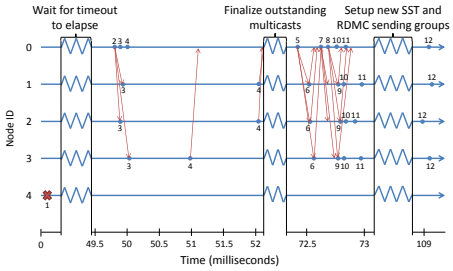


(b) Tracking the events during the multicast of a 200MB message in a heavily-loaded 5-member group.

Figure 2.15: Multicast bandwidth (left), and a detailed event timeline (right).



(a) Timeline for joining an active group.

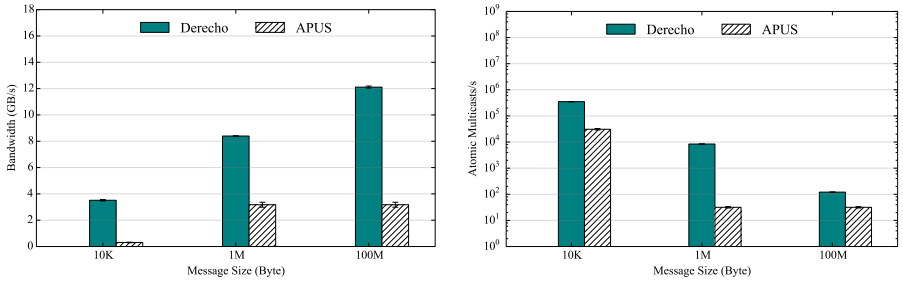


(b) Crash triggers exclusion from an active group.

Figure 2.16: Timeline diagrams for Derecho.

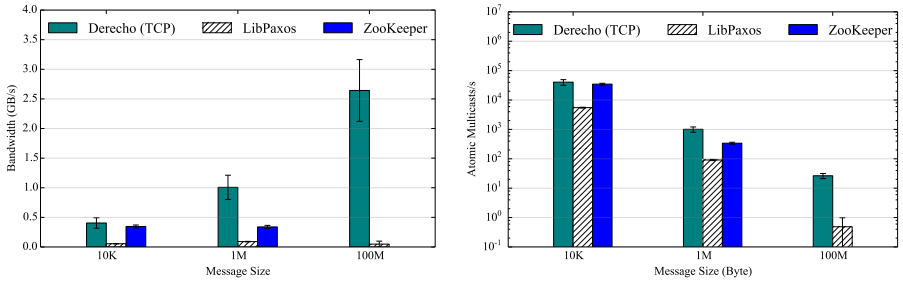
shows handling of a crash; numbering is similar except that here, (4) is the point at which each member wedges.

For small groups sending large messages, the performance-limiting factor involves terminating pending multicasts and setting up the new SST and RDMC sessions, which cannot occur until the new view is determined. This also explains why in Figure 2.15a the disruptive impact of a join or leave grows as a function of the number of active senders: the number of active sends and hence the number of bytes of data in flight depends on the number of active senders. Since we are running at the peak data rate RDMA can sustain, the time to terminate these sends is dominated by the



(a) Throughput (b) Multicast Rate

Figure 2.17: Derecho vs APUS with Three Nodes over 100Gbps InfiniBand.



(a) Throughput (b) Multicast Rate

Figure 2.18: Derecho vs LibPaxos and ZooKeeper with Three Nodes over 100Gbps Ethernet.

amount of data in flight. In experiments with 20MB messages the join and leave disruptions are both much shorter.

2.6.5 *Comparisons with Other Systems*

Using the same cluster on which we evaluated Derecho, we conducted a series of experiments using competing systems: APUS, LibPaxos, ZooKeeper. All three systems were configured to run in their atomic multicast (in-memory) modes. APUS runs on RDMA, hence we configured Derecho to use RDMA for that experiment. LibPaxos and ZooKeeper run purely on TCP/IP, so for those runs, Derecho was configured to map to TCP.

The comparison with APUS can be seen in Figure 2.17. We focused on a 3-member group, but experimented at other sizes as well; within the range considered (3, 5, 7) APUS performance was constant. APUS does not support significantly larger configurations. As seen in these figures, Derecho is faster than APUS across the full range of cases considered. APUS apparently is based on RAFT, which employs the pattern of two-phase commit discussed earlier, and we believe this explains the performance difference.

Comparison of Derecho atomic multicast with the non-durable configurations of LibPaxos and ZooKeeper are seen in Figure 2.18 (Zookeeper does not support 100MB writes, hence that data point is omitted). Again, we use a 3-member group, but saw similar results at other group sizes. LibPaxos employs the Ring Paxos protocol, which is similar to Derecho's protocol but leader-based. Here the underlying knowledge exchange is equivalent to a two-phase commit, but the actual pattern of message passing involves sequential token passing on a ring.

The comparison with ZooKeeper is of interest, because here there is one case (10KB writes) where ZooKeeper and Derecho have very similar performance over TCP. Derecho dominates for larger writes, and of course would be substantially faster over RDMA (refer back to Figure 2.11b). On TCP, the issue is that Derecho is somewhat slow for small writes, hence what we are seeing is not so much that ZooKeeper is exceptionally fast, but rather that the underlying communications technology is not performing terribly well, and both systems are bottlenecked.

More broadly, Derecho's "sweet spot," for which we see its very highest performance, involves large objects, large replication factors, and RDMA

hardware. The existing systems, including APUS, simply do not run in comparable configurations and with similar object sizes.

As a final remark, we should note that were we to compare Derecho's peak RDMA rates for large objects in large groups with the best options for implementing such patterns in the prior systems (for example, by breaking a large object into smaller chunks so that ZooKeeper could handle them), Derecho would be faster by factors of 30x or more. We omit such cases because they raise apples-to-oranges concerns, despite the fact that modern replication scenarios often involve replication of large objects both for fault-tolerance (small numbers of replicas suffice) and for parallel processing (here, large numbers may be needed).

2.6.6 *Additional Experimental Findings*

We also studied API costs. Say that a multicast that sends just a byte array is "uncooked," while a multicast that sends a polymorphic argument list is "cooked." We measured the cost of the cooked RPC framework on simple objects, but discovered it added no statistically significant difference in round-trip time compared to the uncooked case (discounting the initial copy of the RPC arguments into Derecho's sending buffers, a step which can sometimes be avoided for an uncooked send when the data is already in memory, for example after a DMA read from a camera). The primary finding in this work was that copying of any kind can lead to significant delays. Thus applications seeking to get the best possible performance from Derecho should use "zero copy" programming techniques as much as feasible. Within the Derecho API, we avoid copying arguments if the

underlying data format is suitable for direct transmission, and on receipt, we don't create copies of incoming objects so long as the event handler declares its argument to be a C++ `const` reference. Qualifying arguments `const` allows Derecho to re-use the internal buffers into which arguments were received, avoiding any copying in the process of invoking an RPC.

None of those experiments uncovered any signs of trouble: if anything, they support our belief that Derecho can run at extremely large scale, on a wide variety of platforms, and in multi-tenant environments.

2.7 PRIOR WORK

Paxos. While we did not create Derecho as a direct competitor with existing Paxos protocols, it is reasonable to compare our solution with others. As noted in our experimental section, we substantially outperform solutions that run on TCP/IP and are faster than or "tied with" solutions that run on RDMA. Existing RDMA Paxos protocols lack the asynchronous, receiver-batched aspects of our solution. As a result, Derecho exhibits better scalability without exhibiting the form of bursty behavior observed by Marandi and Jalili [Marandi et al., 2014].

With respect to guarantees offered, the prior work on Paxos is obviously relevant to our paper. The most widely cited Paxos paper is the classic Synod protocol [Leslie Lamport, 1998], but the version closest to ours is the virtually synchronous Paxos described by Birman, Malkhi, and van Renesse [K. P. Birman, 2012]. A number of papers suggest ways to derive the classic Paxos protocol with the goal of simplifying understanding of its structure [T. D. Chandra, Griesemer, and Redstone, 2007; Cui et al., 2015;

B. Lampson, 2001; Mazieres, 2007; Prisco, B. W. Lampson, and N. A. Lynch, 1997; Van Renesse and Altinbuken, 2015]

Among software libraries that offer high-speed Paxos, APUS [Wang et al., 2017] has the best performance. APUS implements a state-machine replication protocol related to RAFT, and developed by Mazieres [Mazieres, 2007]. APUS is accessed through a socket extension API, and can replicate any deterministic application that interacts with its external environment through the socket. A group of n members would thus have 1 leader that can initiate updates, and $n-1$ passive replicas that track the leader.

Corfu [Balakrishnan et al., 2012] offers a persistent log, using Paxos to manage the end-of-log pointer,[3] and chain-replication as a data-replication protocol [Renesse and Schneider, 2004]. In the Corfu model, a single log is shared by many applications. An application-layer library interacts with the Corfu service to obtain a slot in the log, then replicates data into that slot. Corfu layers a variety of higher level functionalities over the resulting abstraction.

Round-robin delivery ordering for atomic multicast or Paxos dates to early replication protocols [Chang and Maxemchuk, 1984]. Ring Paxos is implemented in LibPaxos [*LibPaxos: Open-source Paxos* n.d.], and Guerraoui (under the direction of Quéma) has proven a different ring Paxos protocol optimal with respect to its use of unicast datagrams [Guerraoui, Levy, et al., 2010], but Derecho substantially outperforms both. The key innovation is that by re-expressing Paxos using asynchronously-evaluated predicates,

3 Corfu has evolved over time. Early versions offered a single log and leveraged RDMA [Balakrishnan et al., 2012], but the current open-source platform, vCorfu [M. Wei et al., 2017] materializes individualized "views" of the log and has a specialized transactional commit mechanism for applications that perform sets of writes atomically.

we can send all data out-of-band. Section 2.6.5 compares performance of LibPaxos with Derecho.

RAFT [Ongaro and Ousterhout, 2014] is a popular modern Paxos-like protocol; it was created as a replication solution for RamCloud [Ousterhout et al., 2011] and was the basis of the protocol used in APUS. Microsoft's Azure storage fabric uses a version of Paxos [Calder et al., 2011], but does not offer a library API. NetPaxos is a new Paxos protocol that leverages features of SDN networks, though it lacks support for complex, structured applications [Dang et al., 2015]. DARE [Poke and Hoefler, 2015] looks at state machine replication on an RDMA network. RDMA-Paxos is an open-source Paxos implementation running on RDMA [*RDMA-Paxos: Open-source Paxos* n.d.]. NOPaxos [Jialin Li et al., 2016] is an interesting new Paxos protocol that uses the SDN network switch to order concurrent multicasts, but it does not exploit RDMA. None of these libraries can support complex structures with subgroups and shards, durable replicated storage for versioned data, or consistent time-indexed queries. They all perform well, but Derecho still equals or exceeds all published performance measurements.

Atomic multicast. The virtual synchrony model was introduced in the Isis Toolkit in 1985 [K. P. Birman, 1985], and its gbcast protocol is similar to Paxos [Leslie Lamport, 1998]. Modern virtual synchrony multicast systems include JGroups [Ban, 2002] and Vsync [*Vsync reliable multicast library* 2011], but none of these maps communication to RDMA, and all are far slower than Derecho. At the slow network rates common in the past, a major focus was to batch multiple messages into each send [Friedman and Renesse, 1997]. With RDMA, the better form of batching is on the receiver side.

Monotonicity. We are not the first to have exploited asynchronous styles of computing, or to have observed that monotonicity can simplify this form or protocol, although Derecho's use of that insight to optimize atomic multicast and Paxos seems to be a new contribution. Particularly relevant prior work in this area includes Hellerstein's work on logic and lattices for deterministic distributed programming implemented with the Bloom system [Alvaro, Bailis, et al., 2013; Alvaro, Conway, et al., 2011; Conway et al., 2012]. The core result is the *CALM* theorem, which establishes that logically monotonic programs are guaranteed to be consistent. The authors shows that any protocol that does not require distributed synchronization has an asynchronous, monotonic implementation (and conversely, that distributed synchronization requires blocking for message exchange). This accords well with our experience coding Derecho, where the normal mode of the system is asynchronous and monotonic, but epoch (view) changes require blocking for consensus. Other languages have made similar observations; in LVars [Kuper and Newton, 2013], Lindsey Kuper and Ryan Newton also leveraged lattices in building a deterministic-by-construction programming language with monotonic assignment and "threshold" reads, which operate in much the same style as our predicates and triggers in the SST. Christopher Meiklejohn and Peter Van Roy also leverage this technique in Lasp [Meiklejohn and Van Roy, 2015].

DHTs. Transactional key-value stores have become widely popular in support of both the NoSQL and SQL database models. Derecho encourages key-value sharding for scalability, and offers strong consistency for read-only queries that span multiple subgroups or shards. However, at present Derecho lacks much of the functionality found in full-fledged DHT solutions, or DHT-based databases such as FaRM [Dragojević et al., 2014],

HERD [Kalia, Kaminsky, and Andersen, 2014] and Pilaf [Mitchell, Geng, and Jinyang Li, 2013]. Only some of these DHTs support transactions. FaRM offers a key-value API, but one in which multiple fields can be updated atomically; if a reader glimpses data while an update is underway, it reissues the request. DrTM [X. Wei et al., 2015] is similar in design, using RDMA to build a transactional server. Our feeling is that transactions can and should be layered over Derecho, but that the atomicity properties of the core system will be adequate for many purposes and that a full transactional infrastructure brings overheads that some applications would not wish to incur.

2.8 CONCLUSIONS

Derecho is a new software library for creating structured services of the kind found in today's cloud-edge. The system offers a simple but powerful API focused on application structure: application instances running identical code are automatically mapped to subgroups and shards, possibly overlapping, in accordance with developer-provided guidance Each subgroup is backed by a class; interactions between subgroups are made via RMI invocations, making Derecho subgroups effectively replicated actors.

At the core of Derecho is the SST, a replicated table in which every node corresponds to an individual row, which may only be written to by that node. Atop the SST we have built a monotonic language of combinators, enabling consistent, convergent programming despite the relatively weak consistency guarantees of this table. This has allowed us to phrase the core component of Derecho's strong consistency—virtually-synchronous

message delivery—entirely monotonically, allowing Derecho to implement state machine replication at maximum speed.

Derecho can perform updates and read-only queries on disjoint lock-free data paths, employing either TCP (if RDMA is not available) or RDMA for all data movement. Derecho is dramatically faster than comparable packages when running in identical TCP-based configurations, and a further 4x faster when RDMA hardware is available. The key to this performance resides in a design that builds the whole system using steady data flows with minimal locking or delays for round-trip interactions.

3

GALLIFREY

3.1 INTRODUCTION

In the last chapter we learned about Derecho, a framework for building distributed programs with replicated objects in a single datacenter. But modern applications cannot be confined to a single datacenter. The modern internet landscape is filled with *geodistributed* programs: single logical applications split among thousands of machines across the globe. These programs present the illusion of a single available object—be it a Twitter feed, a Facebook timeline, or a Gmail inbox—which is implemented as a constellation of copies, loosely synchronized across perhaps dozens of data centers. This weakly consistent replication became popular due to its performance benefits, but at a significant cost: where objects were once stored on databases offering strong consistency, consistency must now be recovered through the careful effort of application programmers.

Needless to say, it is hard to correctly synchronize replicated objects in this setting. And while past work (Sections 3.8, 2.3) has created an excellent foundation, existing solutions lack modularity and compositionality. Typically, they either fail to provide whole-program guarantees, derive their guarantees from protocols which are too expensive for the wide area (as with Derecho), or rigidly constrain what can be replicated and how it should be replicated (as in the SST core language). Few systems provide

consistency guarantees without forcing the entire program into a single consistency model, and those that do (as in MixT) also lack whole-program correctness guarantees.

This chapter presents Gallifrey, a general-purpose language for distributed programming, whose guiding principles are extensibility, modularity, and flexible consistency. Gallifrey's design encourages engineeringextensible and modular software through the principle of *orthogonal replication*.[1] Under orthogonal replication, the conflict-handling strategy for a replicated object is separated from the implementation of the object itself. Nearly any object can be replicated, yet no object must be replicated.

Gallifrey embodies this principle through a novel language mechanism, *restrictions*. Restrictions refine the interface of a sequential object and provide a *merge function* to resolve concurrent use of allowed methods. Crucially, objects are not tied to a single restriction: programmers may implement many restrictions for a given interface, and may use these restrictions on any object which satisfies this interface. Further, the restrictions on an object may change over time.

Gallifrey combines restrictions with a strong type system to ensure *strong consistency* and *destructive race freedom* by default. Gallifrey leverages the linear, region-based type system presented in chapter 4 to ensure that at most a single thread has access to any given object at a time. Leveraging the isolation guarantees provided by this system, Gallifrey ensures that *restrictions are respected*: that every access to an object guarded by a restriction is made via a method permitted by that restriction. But this guarantee is only as good as the restriction it enforces: it is essential that programmers ensure all permitted operations in a restriction commute, al-

1 The name is inspired by orthogonal persistence [Atkinson and Morrison, 1995].

lowing programs to safely operate against replicated state asynchronously without needing to coordinate during normal execution.

But strong consistency without coordination does not constitute a sufficiently powerful programming model. Real applications evolve, moving through phases in which different operations on data are required. Gallifrey supports these applications through *restriction variants*, an ADT-like mechanism that allows the programmer to share an object under one of several restrictions and use a runtime *match* statement to determine which restriction is currently active. Unlike ADTs, restriction variants also support a *transition* operation which can change the current active restriction, allowing the permitted operations on objects to evolve along with the applications they serve.

We have implemented a prototype of Gallifrey as an extension to the Java language (vers. 7), backed by the Antidote distributed datastore [Akkoorath et al., 2016]. Example Gallifrey programs supported by our prototype can be found in the appendix to this thesis.

We have additionally extended the Gallifrey design far beyond these core implemented features. Chief among these is the idea of allowing principled weakenings of consistency via *provisional operations*. A restriction may specify *provisional operations* that are not required to commute and are therefore, in general, unsafe to use without coordination. Provisional operations can be used only from within explicit *branches*, a new primitive inspired by distributed version control. Branches represent explicit forking of state and serve as the basis for threads, transactions, and speculative execution. Branches and provisional operations combine to allow speculative execution; provisional methods executed within a branch remain isolated in that branch until it is explicitly merged, either synchronously

```
interface Library {
  int numItems();

  bool in_collection(preserves Item i, preserves String col)

  bool in_library(preserves Item i)

  isolated Set[Item] getItems(preserves String col)

  void addCollection(consumes String col)

  void addItem(consumes Item i, preserves String col)

  // creates "into" if it does not exist, has no effect if "from" does
      not exist
  void mergeCollection(preserves String from, preserves
      String into)

}
```

Figure 3.1: Library interface. `preserves` and `isolated` decorate arguments and returns which are not stored by the library, while `consumes` decorates arguments which are.

or asynchronously. When merged synchronously, branches have the semantics of optimistic transactions, and thus sacrifice no consistency; when merged asynchronously, branches have a weakly consistent semantics, as provisional operations contained within a branch may conflict with other concurrent operations. To compensate for such conflicts, programmers provide a callback as a *contingency* to be executed if a conflict does occur.

3.2 A RUNNING EXAMPLE

To better understand the difficulties of programming with replicated objects and how Gallifrey makes this task easier, we introduce a running

example. Consider a "library" object (Figure 3.1) that maintains a set of items grouped by collections—for example, a set of books collected under "Programming Languages" might include *Structure and Interpretation of Computer Programs* [Abelson and Sussman, 1996] and *Types and Programming Languages* [Pierce, 2002]. Alice and Bob use this library object to keep citations for a paper they are writing together. Like many academics in the pre-pandemic era, Alice and Bob find themselves frequently traveling to conferences, working on their bibliography on the go—including in places with limited internet connectivity. Their bibliography application must allow them to continue working while disconnected. Now, suppose Alice adds a book to the collection, *How to Design Programs* [Felleisen et al., 2001], while at the same time Bob merges the "Programming Languages" collection itself into some default collection. To what state of the library should Alice and Bob's devices both eventually converge?

There are two strategies for responding to such irreconcilable conflicts. One is *prevention*: restrict concurrent execution of operations that might conflict. For example, Alice and Bob might agree to not remove collections from the library so that either of them can add books and query the library safely. The second is *restoration*: provide a way to safely merge conflicting operations.[2] Alice and Bob can agree on a restorative strategy by allowing book additions and collection merges, but *without* allowing users to query the status of collections at all. This restriction prevents either Alice or Bob from acting on unstable knowledge of the collection; While Bob may remove a collection under this restorative strategy, he must do so without any assurances that the collection in question was even in the library in

2 Indigo [Balegas et al., 2015] makes a similar distinction between *conflict avoidance* and *conflict resolution*.

the first place. Later, Alice and Bob can achieve *consensus* on the state of the library, revealing the fate of its collections.

Now suppose Alice gets on a plane and wants to see what books are in the library. Without being connected to Bob, Alice can't be sure that the list of books she's seeing contains all the books in the library; after all, Bob could have added more books while Alice wasn't looking. Alice might be fine with this. Perhaps she was only interested in checking if the library was at least a certain size, or contained at least a certain set of texts. This she can do safely even without Bob, since Alice and Bob both agreed not to remove items from the library. But if Alice wishes to perform some action which may invalidate Bob's observations, or make some observation which Bob's actions could invalidate, she cannot do so on the plane. Instead, she must wait until she lands, reach consensus with Bob on the state of the object, and then *transition* to a new agreement with Bob under which Alice's operations will be permitted (i.e. Bob will not perform any operations with which they may conflict).

Gallifrey's programming model is designed for this challenging setting.

3.3 RESTRICTIONS FOR SHARED OBJECTS

The primary purpose of Gallifrey—safely sharing objects via asynchronous replication—is enabled by *restrictions*. Restrictions represent the conflict-handling strategies for replicated objects. Restrictions are a part of the type of a replicated object, and Gallifrey uses them at compile time to ensure that all replicas agree on a conflict-handling strategy. Syntactically,

```
restriction AddOnly for Library {
  allows addItem;
  allows as test in_collection;
  allows as test in_library;
  test sizeAtLeast(int n) { return numItems() >= n; }
}

restriction AddWins for Library {
  allows addItem;
  allows mergeCollection;
  allows as test in_library;
  test sizeAtLeast(int n) { return numItems() >= n; }

  merge (addItem(Item i, String ic) l, mergeCollection(
      String oc, String nc) r) {
    order l < r;
  }
}
```

Figure 3.2: Restrictions for `Library` interface.

an object declared with type `shared[R] T` is of class `T` and is shared under a restriction `R`.

Restrictions are defined against a specific interface. For example, Figure 3.2 shows two possible restrictions for library objects: `AddOnly`, which only allows `addItem` operations and testing if an item is in a collection, and `AddWins`, which allows `addItem` and `mergeCollection` but cannot check if an item is in any particular collection. These correspond to the two conflict-handling strategies in Section 3.2. A restriction consists of the following parts:

Interface refinements. Restrictions specify exactly which operations of an interface are allowed under them. Any operation not specified in a restriction cannot be executed under it, thus allowing for preventative conflict-handling strategies. For example, in Figure 3.2 `AddOnly` prohibits

collection removals. Allowed operations in a restriction must never interfere; doing so results in undefined behavior, and can be statically prevented by our proposed pre- and postcondition analysis.

Merge functions. Restrictions include *merge functions* to handle any conflicts that may arise when two operations execute concurrently, thus allowing for restorative conflict-handling strategies. Merge functions pattern-match over pairs of operations and their arguments, and then dictate in which order those operations should appear to have occurred. Merge functions are only invoked on pairs of operations which are not causally related; two operations which *are* causally related will always appear to occur in their causal order. For example, in Figure 3.2 `AddWins` contains a merge function requiring that all collection merges occur after any concurrent additions to those collections. Merge functions are free to observe, compute on, or modify the arguments to the operations they are ordering.

Monotonic tests. Because updates to replicated objects can be reordered, reads of the object's state before convergence can vary across replicas. Thus, reading a replicated object's state directly is usually eschewed: instead, a special class of reads, found in programming models such as LVars (*threshold reads*) and Lasp (*monotonic reads*), is defined [Kuper and Newton, 2013; Meiklejohn and Van Roy, 2015]. Restrictions provide a similar functionality with *monotonic tests*: boolean expressions whose value is guaranteed to remain true once it becomes true, no matter what further operations are received by the replica. With this property, monotonic tests can be used for *triggers*, code whose execution is blocked until a monotonic test becomes true. These monotonic tests and triggers are closely related to the predicates and triggers found in Derecho's SST core language. For example, in Figure 3.2 the `AddWins` and `AddOnly` restrictions allow

$$\tau <: \texttt{shared}[R]\ \tau$$

SHARED-REPLICATES

$$\frac{\Gamma; \mathcal{H} \vdash e : \ell\ \texttt{shared}[R]\ \tau \dashv \Gamma'; \mathcal{H}'}{\Gamma; \mathcal{H} \vdash e : \ell'\ \texttt{shared}[R]\ \tau \dashv \Gamma'; \mathcal{H}', \ell' \langle\ \rangle}$$

Figure 3.3: Extension to chapter 4's isolation typing for shared objects.

programmers to invoke the `in_library` method as a test. Both also feature `sizeAtLeast`, a test constructed directly in the restriction which internally uses the otherwise-not-permitted `numItems` method to test whether the number of items in the library has passed some threshold. If Alice (from Section 3.2) is worried that the library is getting too big, then this test can be used to inform her that the library is bigger than some threshold size.

3.3.1 *Safety Guarantees via Isolation Typing*

Importantly, restrictions offer the following type-safety guarantee:

- No object can perform an operation forbidden by the restriction under which it is shared.

Correct restrictions—those which do not permit conflicting operations—must also ensure the following property:

- Monotonic tests cannot be invalidated: once their value is true, their value will always be true afterward *until replicas explicitly coordinate*.

Taken together, these properties provide a strong safety result: a program with correct restrictions always enjoys strong consistency.

Gallifrey derives its type-safety guarantee from its use of isolation typing as presented in chapter 4. Chapter 4 introduces a static type system

which ensures that a reference is *isolated*: that any objects reachable from a reference may only be reached via that reference. By treating shared references as isolated, this system in turn ensures that each shared object is only accessible via its correctly-restricted interface. Chapter 4's system is organized around the idea of regions; all objects belong to some region, and the type system statically tracks the region to which all objects belong (labeled ℓ) and how the objects in different regions relate.

There are only two Gallifrey-specific extensions to chapter 4's system required to implement shared objects. The first is to introduce shared references as an explicit type, and treat them as a supertype of unshared references. The second is to declare that a shared reference can live in any region; this reflects the fact that these shared references are always allowed to cross regions. These rules are reflected in figure 3.3.

Gallifrey also requires that restrictions do not expose methods which conflict; but care must be taken in determining conflicting methods. Consider for example a restriction which allows a single method `void m(preserves Box<T> t)`. Despite the fact that this method has no return result, it is *still possible* that this method could self-conflict if it mutates its argument `t`. This method could be implemented as a combination setter/getter with `t` as its input and output parameter:

```
class Container<T> {
  T t;
  void m(preserves Box<T> t){
    T tmp = this.t;
    this.t = t.get().clone();
    t.set(tmp.clone());
  }
}
```

Calling `m` here reveals the *exact value* of a field which may be mutated by concurrent calls to `m`. We have effectively implemented a read-write register;

using these with observationally strong consistency requires linearizability, which cannot be implemented during disconnection—and restrictions must function during disconnection. It is therefore never safe to allow `m` in a restriction.

3.4 TRANSITIONING RESTRICTIONS WITH RESTRICTION VARIANTS

One might find shared object restrictions *too* restrictive: single restrictions are essentially static contracts, and ban certain operations for so long as they are in force. Prior work [Magrino et al., 2019; Roy et al., 2015; Whittaker and Hellerstein, 2018] has shown that loosely synchronized replicas can eschew coordination for most operations, and then coordinate only to safely change established invariants. Chapter 2 contains a concrete example of this: the SST from Derecho. Using the SST, programmers can mutate shared monotonic datatypes, much as Gallifrey allows programmers to share arbitrary objects under monotonic restrictions. But Derecho also needs to violate monotonicity in order to handle a *view change*, triggered by node failure or group reconfiguration. During view change, Derecho achieves consensus on the exact values in the SST, *transitioning* to a new set of monotonic datastructures active in the new view.

Taking a cue from this work, we propose two additional features for restrictions. The first is *reclamation*: a thread may attempt to *reclaim* a shared object as a local object, removing its restriction and eliminating its ability to be replicated. This parallels the activation of a leader in Derecho's view-change protocols. This operation is only possible if no other replicas

of this object exist; the system must reach consensus on this fact before a reclamation can proceed.

More usefully, we also introduce a mechanism to *transition* shared objects across restrictions. The strict separation of object implementations and conflict resolution strategies allows programs to dynamically transition between restrictions, changing the conflict-handling strategy of shared objects over time. At the point of transition, all replicas reach consensus on the true value of an object, allowing a new restriction to be safely applied. One can view Derecho's view-change protocol as a special case of this transition operation, reaching consensus on and changing the exposed rows of the SST.

3.4.1 *Restriction Variants*

To enable transitions we first introduce a new kind of restriction: *restriction variants*. A *restriction variant* allows programmers to share an object under one of several possible concrete restrictions. An example specification of a restriction variant is found in Figure 3.4 on line 5; here we see that restriction variants are specified syntactically as a name given to a list of concrete restrictions separated by vertical bars.

Restriction variants work much like sealed classes from Kotlin. As with a sealed class, the variant acts as an "abstract" restriction, allowing none of the restricted class's operations. But because we statically know all possible "concrete" restrictions a variant-restricted object may have at runtime, we can explicitly `match` on a variant-restricted object to discover which concrete restriction is active on that object. In the arms of this match we receive

```
restriction ReadOnly for Library {
 allows getItems;
}

restriction Threshold = AddWins | ReadOnly

class LibraryClient {
 shared[Threshold] Library library;
 shared[Messaging] User user;

 public LibraryClient(shared[Threshold] Library lib,
                      shared[Messaging] User u) {
  library = lib;
  user = u;
  match_restriction library with
  | shared[Threshold::AddWins] Library awlib {
   changeRestriction(awlib); }
  | shared[Threshold::ReadOnly] Library rolib { } }

 void addItem(consumes Item item, consumes String collection) {
  match_restriction library with
  | shared[Threshold::AddWins] Library awlib {
   awlib.addItem(item, collection); }
  | shared[Threshold::ReadOnly] Library rolib {
   throw ClientException("Library is read only!"); } }

 void mergeCollection(preserves String from, preserves String
     into) {
  match_restriction library with
  | shared[Threshold::AddWins] Library awlib {
   awlib.mergeCollection(from, into); }
  | shared[Threshold::ReadOnly] Library rolib {
   throw ClientException("Library is read only!"); } }

 isolated Set[Item] getItems(preserves String collection) {
  match_restriction library with
  | shared[Threshold::AddWins] Library awlib {
   throw ClientException("Library must be read only!"); }
  | shared[Threshold::ReadOnly] Library rolib {
   return rolib.getItems(collection); } }

 void changeRestriction(shared[Threshold::AddWins] Library awlib)
     {...} }
```

Figure 3.4: Client that uses a shared library object.

a more precise reference allowing all operations permitted by the active concrete restriction. The reference we receive in this arm is not, however, a normal concrete-shared reference, but one specialized to the restriction variant on which we matched. An example of such a match can be found in Figure 3.4 on line 21; here, we match on a library shared under a variant permitting both `AddWins` and `ReadOnly` concrete restrictions, with one arm for each concrete restriction. In the first branch, the reference `awlib` is bound to a `shared[Threshold::AddWins]` variant of `library`. In the second, the reference `rolib` is bound to a `shared[Threshold::ReadOnly]` variant of `library`.

The semantics of holding a shared reference for a specific arm of a restriction variant (e.g. a `shared[Threshold:AddWins]` reference) are quite different from those of sealed classes due to our desire to support *transitions*. The goal of transitioning is to change the concrete restriction of an existing variant-shared object at runtime. But a new concrete restriction may allow new operations on the shared object, which may conflict with those allowed under existing specific variant references. So long as a specific `shared[Threshold:AddWins]` reference is valid somewhere in the system, it would be incorrect to allow any operations which could conflict with `AddWins`—effectively locking the object to the `AddWins` concrete restriction. To prevent objects from being permanently locked to a specific concrete restriction, we limit the lifetime of variant-provided concrete restrictions to the duration of the match statements which produce them. These specific variant restrictions are *implicitly nulled* when the match ends, preventing programmers from relying on them past the end of the match. This is also why we have chosen to syntactically link these references to their original variant restriction; by creating a different syntactic class for

these more-ephemeral restrictions, we allow the programmer to only worry about implicit nullability on references which will actually experience it.

This reliance on implicit null may be a cause for concern—after all, one of our goals in chapter 4 is to *avoid* any use of implicit null. We rely on it here to increase the expressive power of shared references. We want to allow programmers to send variant-shared objects stored under a concrete restriction to other threads or processes, without relying on structured parallelism to ensure these objects have limited lifetimes. Determining a usable static mechanism which achieve this without introducing significant programmer-facing complexity is an exciting avenue for future work.

3.4.2 *Transitioning Restrictions*

We now present the details of `transition`. The primitive operation `transition()` creates a *request* to transition an object shared under a restriction variant to one of its constituent restrictions. After any replica requests a transition, Gallifrey's runtime attempts to reach consensus on both the value of the object and the new requested transition at all replicas, waiting until any relevant `match` statements have terminated and blocking any future `match` statements until consensus can be achieved or a timeout occurs. If a timeout occurs, the `transition` does not take effect, leaving the object under its current restriction. Even if a transition *does* occur, there is every possibility that *another* transition may occur immediately afterwards, taking the object to an unexpected concrete restriction. For this reason, programmers can *never assume* that a `transition` has succeeded, but rather must `match` on the object to determine its current active restriction.

```
void changeRestriction(shared[AddWins] Library awlib){
 thread (awlib, user) {
  when (awlib.sizeAtLeast(100)) {
4   user.sendMessage("Library is too big!");
    transition(awlib, ReadOnly);
  }
 }
}
```

Figure 3.5: Using a trigger to transition restrictions.

For an example of transitions between restrictions, consider Figure 3.5. The `LibraryClient` constructor calls `changeRestriction`, which creates a thread with a new replica of the library object that adds a trigger to transition its library object to `ReadOnly` when the library reaches a certain size using the `sizeAtLeast` test defined in `AddWins`. The `addItem` and `mergeCollection` methods match on the current restriction of the library to ensure it is `AddWins`; otherwise the methods throw an exception. The `getItems` method does something similar for the `ReadOnly` restriction.

3.5 THE GALLIFREY IMPLEMENTATION

We have implemented a Gallifrey prototype as an extension of Java 1.7 using the Polyglot extensible compiler framework [Nystrom, Clarkson, and A. C. Myers, 2003], backed by the distributed object store Antidote [Akkoorath et al., 2016]. Our prototype does not realize the full promise of the Gallifrey programming model; it does not incorporate the type system proposed in chapter 4 (opting for a simpler unique-pointer model similar to that of C++), and all features described in the following "Design"

sections have not been implemented. Nonetheless, it provides an excellent research platform via which to experiment with Gallifrey.

3.5.1 *The System: Using Antidote for shared Objects*

The architecture of Gallifrey's object sharing distributed system is reasonably straightforward. The Gallifrey compiler frontend translates the implicit creation of shared objects to the explicit creation of an object stored in the Antidote distributed system [Akkoorath et al., 2016]. Antidote provides the abstraction of an object store offering causal consistency. Each object within Antidote is an instance of some Convergent Replicated Data Type (CRDT) [Shapiro, 2017] conforming to Antidote's published CRDT interface. While Antidote is extensible and allows user-provided CRDTs, it cannot change the set of CRDT types available at runtime; adding a new type of CRDT requires a restart of the Antidote system. To compensate for this, we choose to associate all shared objects in Gallifrey with a single Antidote CRDT type representing a serialized Java object "snapshot" and a partially-ordered log of operations to apply to this object. When a Gallifrey object is initially shared, a snapshot of that object is taken and a new, empty operation log is created for the object. This pair of snapshot and log is then sent to Antidote. Subsequent mutations to the object are not applied directly to the snapshot, but rather stored in the object's operation log.

Within Antidote, each replica of an object is managed by a private Java "backend" process. This process manages the log of operations pending for each shared object, applying those operations to the snapshot as frequently

as possible, creating a new snapshot and removing the applied operations from the operation log. In order to maintain consistency, it is imperative that all operations appear to occur in the same order at all replicas. This in turn means the backend must totally order pending operations, and cannot flush an operation until all its predecessors have arrived. Determining a total order is complicated by the realities of causal replication; if two disconnected replicas each receive an operation, they cannot immediately flush it to their snapshot as each replica knows that there may exist some other operation which should be serialized before the operation they have received.

We solve the ordering problem by combining vector clocks with user-provided `merge` functions. Antidote itself maintains a single global vector clock shared by all CRDTs under its care; we leverage this mechanism to provide a vector clock timestamp to both snapshots and operations. Antidote also provides a notion of a "global minimum time": an artificial vector clock value guaranteed to be causally before (or the same as) the local time at every replica. As Antidote guarantees causal replication, it is impossible for a replica to learn of a new global minimum time before that replica has received all operations which occurred before that new global minimum time. This global minimum time in turn allows the definition of a "global minimum prefix" of an operation log: the set of operations which occur before the global minimum time. Causality ensures that the global minimum prefix is complete: there are no "missing" operations which may occur between any two events in this prefix.

The global minimum prefix still requires a total order. This total order is generated from the user-provided `merge` functions. Whenever two operations are unordered in the global minimum prefix, their order is

determined by their object's restriction's `merge` function when available, or lexical ordering on vector clocks when a `merge` is unavailable. Combining merge functions, lexical ordering, and causal ordering in this way generates a total order; this is the order in which operations in the global minimum prefix are applied to the snapshot.

Only one puzzle remains: how to communicate observations of shared objects back to the observer. To assist with this, we require that methods allowed in restrictions (for this prototype) are divided up into a set of "commands" and a set of "queries." Commands are mutations which can neither return a result nor mutate their arguments; Queries may return results and mutate arguments but may not mutate their receiver. Commands are appended to the shared object's operations log: they are represented as a serialized method name and arguments, and are managed by Antidote. Queries are less straightforward, as they must reflect the results of any pending commands in the operation log. To handle these, our prototype creates a temporary copy of the Antidote-managed snapshot and log, applies all the operations in the log to this temporary copy, and return the result of the query on this fast-forwarded copy.

3.5.2 *Transitions and Consensus*

While Antidote is able to manage the CRDT-like aspects of Gallifrey's shared objects, it is not a good platform match for its synchronous transition mechanism. To implement matching and transitions for restriction variants, we introduce a new centralized service which manages the current concrete restriction for all objects shared under a restriction variant.

On a match statement, a Gallifrey client must contact this central service to determine which current restriction is active on their variant-shared object.[3] For the duration of the match, the client holds a read lock on this concrete restriction; all transition requests will be blocked (or rejected via timeout) while any client is currently in a match arm. This ensures that concrete-restriction-tagged shared references will remain valid for the duration of each match.

Transition requests are also processed via this central service. To request a transition, the client first contacts the central service to determine if any concrete restrictions are active for the transitioning object. If not, it then acquires a write lock on the concrete restriction for this object, blocking any other clients from entering match statements. The central service in turn contacts all replicas of the object, flushing pending operations and waiting for all in-flight operations to be received at every replica. Once all replicas have converged, the central service allows the transition to proceed, setting the current concrete restriction accordingly. At this point all locks are released, and clients are free to enter matches once again.

3.5.3 *The Compiler: Extending Java 1.7 to Handle Gallifrey*

The Gallifrey compiler is implemented in a Polyglot extension to Java 1.7, extended to handle restrictions, restriction variants, merge functions, and tests. In all cases, the objects and interfaces described here guard an instance of a generic `SharedObject` type, which supports a reflection-like

3 Other solutions, such as allowing clients to take out leases on their concrete restrictions, may yield improved performance; the ultimate design of this component is heavily influenced by the deployment domain of Gallifrey and as such is best specified as a pluggable component.

API for invoking methods and handles all replication and communication activities via the mechanisms described in the previous subsections.

Restrictions Our translation treats restrictions as extra interfaces implemented by an object. When compiling a class, the project is searched for all restrictions that mention this class. The compiler then generates interfaces corresponding to these restrictions, containing the methods that the restriction `allows`. The compiler then adjusts the restricted class to explicitly `implement` all of its restrictions. During typechecking, a `shared[R] T` is *not* treated as a direct supertype of `T`; programmers must explicitly call a special `move` function to transform an instance of `T` into an instance of `shared[R] T`. We then can use this explicit codepoint to insert the relevant Antidote operations. While this `move` call can be entirely inferred in principle, it is not inferred by the current compiler.

Restriction Variants. Our translation for restriction variants is necessarily quite a bit different from our translation for plain concrete restrictions due to Restriction Variants' ability to transition. First, the compiler must synthesize separate variant-specific restriction interfaces corresponding to each arm of this variant. These interfaces all extend a marker `holder` interface for this variant, and are otherwise identical to those of the matching concrete restriction. The Restriction Variant itself is represented as a concrete class with a single method `transition(Class<?>)` and a single field holding the current active concrete restriction. Unlike with plain restrictions, however, the interfaces corresponding to arms of a restriction variant are *not* extended by the concrete class they restrict; rather, they are implemented by a concrete class which extends the *restriction variant class* and wraps the corresponding concrete restriction, forwarding its methods. This encoding correctly ensures that a shared reference restricted via an

arm of a restriction variant cannot be confused with one restricted by a concrete restriction, even if both references are based on the same concrete restriction.

Match Statements. The difficulty in translating `match` statements does not lie in the actual pattern match itself — that is easily handled by translating the `match` cases to a chain of `if` statements, which rely on Java's `instanceof` operator to determine if the `holder` field of the restriction variant in question contains a concrete restriction of the target type. Rather, the difficulty is in ensuring that no `transition` can occur while those references are active, and that the references exposed via the `match` are correctly nulled out at the end of this `match`. The former is accomplished through the read-lock logic discussed in the previous subsection. To accomplish the latter, the runtime creates a shallow copy of the restriction variant object for each match branch, allowing the `holder` field to alias that of the original variant object's `holder` field. Consider for example the following code:

```
shared[Foo] Baz obj = ...;
match_restriction obj with
  | shared[Foo::Bar] Baz bobj {bobj.bop(); }
```

This code roughly translates as follows:

```
Foo obj = ...;
final Foo_Bar bobj;
try {
  obj.read_lock();
  if (obj.holder() instanceof Bar){
    bobj = new Foo_Bar_impl(obj.holder());
    bobj.bop();
  }
} finally {
  bobj.clear_holder();
  obj.read_unlock();
}
```

For this fragment, the compiler generates a `final` object to store the reference for the match arm, ensuring that errant programmer assignments cannot invalidate the nulling-out logic. In a try block, the code first acquires the read lock on the matching object (blocking transitions), and then determines if the wrapped concrete restriction is an instance of the desired concrete restriction. If it is, it is unwrapped into a new object corresponding to the matching arm of the restriction variant. Finally, a `finally` block clears the `holder` field of the match-limited object, ensuring subsequent uses of it receive `NullPointerExceptions`, and the lock is released.

Tests. Our test translation requires two things: first, that a programmer may not call a test method outside of a `when` block, and second, that programmers may add test methods at the level of restrictions. To accomplish this, our compiler migrates test methods back to the restricted class, and mangles their names to include the restriction from which they came. It then translates method calls within `when` blocks via the same name-mangling logic. `When` blocks themselves are translated as a while loop on the condition, blocking until it becomes true, followed by the body.

3.6 DESIGN: THE PROMISE OF PRE- AND POST-CONDITIONS

Writing correct programmer-provided restrictions can indeed be difficult. It is possible to eliminate much of this difficulty and *enforce* that restrictions are correct by requiring users provide preconditions and postconditions, and analyzing these for conflicts.

Through the Java Markup Language (JML) [Badros, 2000], the Java ecosystem has long provided support for annotating methods with pre-

and post-conditions. These are written as logical formulas over uninterpreted functions, and appear in Java annotations in comments above method declarations. In its modern incarnation as OpenJML [Cok, 2011], this software has matured into an SMT-based verification solution for all of Java 1.8. This verification attempts to establish *functional correctness*: whether the implementation of a Java method matches the specification provided in the comments. We need it to do more: to determine not just that all methods of an interface are implemented correctly, but to determine which methods in an interface may be safely called concurrently or during disconnection.

To accomplish this, we take inspiration from Indigo [Balegas et al., 2015] and allow our JML annotations to include abstract predicates and specifically-marked read operations defined in the interface. Abstract predicates do not have a concrete definition; they are asserted directly in pre- or postconditions in order to describe the assumptions and effects of an operation over an object's state. Including read operations in the language of postconditions allows us to connect these postconditions with the state of the object, describing how subsequent reads will be affected by an operation. These annotations allow the detection of conflicts that arise from concurrent operations—for example, when the postcondition of one operation violates the precondition of another, or when two operations have conflicting postconditions. Thus the type checker can determine whether any mutation threatens to invalidate a test or observation, and only permit restrictions which do not contain such *conflicting operations*. Like Indigo and OpenJML, we use an SMT solver to verify that the pre- and postcondition annotations on operations are consistent with our desired safety guarantees [Balegas et al., 2015].

```
interface Library {
  int numItems();

  isolated Set[Item] getItems(preserves String col)
    requires collection(col);

  void addCollection(consumes String col)
    ensures collection(col);

  void addItem(consumes Item i, consumes String col)
    requires collection(col)
    ensures next(numItems()) >= numItems();

  // also moves items in col to a default collection
  void removeCollection(preserves String col)
    ensures !collection(col) && (next(numItems()) == numItems
        ());
}
```

Figure 3.6: Simplified and annotated Library interface. `requires` and `ensures` refer to pre- and postconditions respectively. `collection` is an abstract predicate indicating the presence of a collection in the library.

Consider the simplified annotated `Library` interface in Figure 3.6. Here, `addItem` adds an item to an existing collection if it is not already in the collection, so its postcondition says that the return value of `numItems` after invocation of `addItem` (i.e. `next(numItems())`) is at least the return value of `numItems` before invocation—the number of items in the library remains the same or it increases by one. Meanwhile, `removeCollection` removes a collection from the library without removing the items in it from the library, instead moving orphaned items (those not in any other collection) to a default collection. Since the postcondition of `removeCollection` violates the precondition of `addItem` when their arguments reference the same collection, the concurrent operations conflict. Note that all operations have postconditions that do not decrease the value of `numItems()`, allowing us to verify that `numItems()` is monotonic with respect to all library mutations.

3.7 DESIGN: WEAKENING CONSISTENCY WITH PROVISIONAL OPERATIONS

In this and subsequent sections we discuss aspects of the Gallifrey language that we have designed, but have not substantially evaluated or implemented. Nevertheless, we consider these ideas to be a key contribution of the Gallifrey effort.

3.7.1 *Provisionality via Branches and Contingencies*

Section 3.3 discussed Gallifrey's use of restrictions to guarantee strong consistency and whole-program convergence. But these guarantees re-

quired restrictions in which no operations conflicted, with only the highly-synchronous `transition` method available to build objects for which calling conflicting operations is desired. These limitations are impractical for many common programs; sometimes programs may need to read *and* write a shared object, without stopping for consensus between operations.

Recall that in section 3.2 Alice and Bob are collaborating to maintain a library. Let us imagine a scenario in which Alice gets on a plane and wants to check if a certain book is already present in the library, adding it to some collection if it is not. Under the mechanisms we have presented so far, this is impossible: adding books to the library is permitted but removing them is not, so the library monotonically grows. Under this setting one may safely condition on action on the *presence* of a book in the library, as this is guaranteed to be maintained; but Alice wants to condition an action on the *absence* of a book from the library. This cannot be done soundly; a book Alice observes to be absent may have been added to by Bob while Alice was taking off.

To allow this, Gallifrey permits restriction to *provisionally* allow methods. Provisional methods leave open the possibility of conflicts; in exchange, there are no limitations on what a provisional method can do. These methods are executed optimistically, allowing users to continue operating against replicated state without stopping for consensus. Because provisional methods can conflict, their effects (and any program statements which depend on them) are not guaranteed to be consistent; because of that, any provisional method allowed in a restriction must be *contingent* on the absence of a conflict, and must be associated with a *contingency callback* to be invoked if a conflict does occur.

```
class ItemsStale {...}

restriction AddAndCheck for Library {
  allows addItem;
  allows as test in_collection;
  allows as test in_library;
  test sizeAtLeast(int n) { return numItems() >= n; }
  allows getItems contingent ItemsStale

  merge(addItem(Item i, String col) l,
        getItems() r -> Set[Item] s){
    if (!s.contains(i)){
      reject r with ItemsStale();
    } else order l < r
  }
}
```

Figure 3.7: A restriction with a provisional method.

In figure 3.7, we see a restriction `AddAndCheck` which extends the previous `AddOnly` restriction with the *provisional* operation `getItems`. The keyword `contingent` serves to both mark this operation as provisional and name `ItemsStale` as the type that will be used to communicate with required contingency callbacks. Under this restriction, calls directly to `getItems` will be allowed—even though the value returned by `getItems` is not guaranteed to be consistent.

But one cannot simply execute potentially conflicting actions without acknowledging the significant inconsistency invited by doing so. To partially recover from this, Gallifrey pairs every provisional method with a *contingency*: a named callback intended to recover from—or at least apologize for—any consistency error resulting from using a provisional method. Contingencies are invoked directly from the merge function for the associated restriction, as seen in figure 3.7, and so can receive any necessary

```
void addIfAbsent(shared[AddAndCheck] Library aclib,
                 consumes Item book,
                 preserves String collection){
  Branch tok = branch(aclib, collection) {
    if (!aclib.getItems().contains(book))
    aclib.addItem(book,collection);
  };
  tok.pull(ItemsStale is => {
    user.sendMsg("Read was stale: " + is.observed);
  }, Success succ => {
    user.sendMsg("Valid read of" + succ.observed);
  });
}
```

Figure 3.8: Using branches for a provisional operation with contingency.

information from the merge. To handle this, we extend the capabilities of the merge function, allowing it to explicitly `reject` an operation whose consistency cannot be rescued via re-ordering.

As a simple example of the use of these features, consider the function defined in Figure 3.8. Here, we've implemented our extended example in which Alice tries to add a book to the library based on its *provisional* absence from another collection. This leaves open the possibility that a merge function would reject this operation. To compensate, Alice registers a contingency callback, which sends an error message indicating the observation failed to include something (Figure 3.8, line 9).

3.7.2 *Branches*

Using provisional methods and contingencies raises important semantic questions. After the invocation of a provisional method on a shared object, are all subsequent uses of this object also provisional? If a provisional

observation from an object flows to other values in the program, should those values also be considered provisional? What if that flow reaches different, unrelated shared objects? And precisely where is the right place to register a contingency callback—close to the provisional invocation, or close to the eventual visible use of its result?

In Gallifrey, the key to answering all these questions is a new mechanism called *branches*. Branches exist to contain provisionality; like their namesake in the world of version control, every branch possesses its own fork of state, isolated from external mutations until it is *merged* back into its parent. Programmers may enter and exit ("check out") in-progress branches, spawn sub-branches, and freely choose to discard or merge branches. When a provisional operation occurs within a branch, then the *entire branch* is considered provisional; any code that executes after a provisional operation may be tainted by that provisional operation, and so inherits all of its potential for conflict. Helpfully, branches also allow deferring the point at which contingencies are required. Because branches are strongly isolated from the remainder of the system, any potential conflicts are safely contained within the branch; the only point at which these conflicts become visible is when the branch attempts to merge with the outside world. At this point we require that programmers supply contingencies.

Syntactically, branches are created by the syntax `branch(args...){body}`, as in Figure 3.8. The `args...` is a list of objects that the branch `consumes` (as in the function argument decorator) and now owns, and which are available within the branch's body. When a shared reference first passes into a branch, a new replica is made for the branch. The branch's body is executed immediately, after which control proceeds with the statement immediately following the branch. The `branch` construct also returns a

token via which programmers can interact with the branch. This token is linear; it cannot be aliased, and it must be either merged or aborted before it goes out of scope.

A typical use of the branch's token, as seen in Figure 3.8 on line 8, is to optimistically merge the branch into the calling context via `pull`. This construct immediately merges everything in the branch with `pull`'s calling context, leaving open the potential for conflict with other, as-yet-unmerged branches. Because of this potential for future conflict, users must provide a set of contingency callbacks covering all provisional behavior that occurred on this branch. These callbacks are intended as a means to repair any damage done in the case of a consistency violation caused by any conflict.

To avoid the possibility of conflict, Gallifrey's branches also support a synchronous `commit` operation. Like `transition`, `commit` blocks until a consensus can be reached among replicas, determining which provisional operations are consistent with global state, and which, having been found in conflict with already accepted operations, should be rejected by the system. After commit, all effects from within the branch become visible to the wider system; operations rejected due to conflict are re-executed against consistent state, with the new results replacing the old. With `commit`, branches have the semantics of strictly serializable transactions. Branches operate on an isolated snapshot of state, apply the effect of all their operations, verify that their snapshot remains consistent with the system at large, and re-execute their operations if not.

The branch token can be used for more advanced features as well. With *`token`*`.abort()`, programmers can explicitly abandon the branch. With *`token`*`.peek`, programmers can steal a reference to the branch's state *without*

```
void atm_withdraw(shared[All] Account acct, unique Integer
    amnt) {
 Branch tok = branch(acct,amnt){
  Integer withdrawn_amnt = acct.withdraw(amnt);
  Double percent = withdrawn_amnt / acct.balance();
 };
 if (tok.peek[percent] <= 0.25){
  tok.pull(Overdraft amnt => { charge_overdraft(acct) },
         UpdatedBalance => {/* ignore */});
 } else tok.commit();
}
```

Figure 3.9: More advanced features of branches.

first merging the branch, and *without* needing to supply contingencies so long as the result of `peek` does not influence any visible actions outside the branch.

These features are illustrated in Figure 3.9. This figure introduces the example of withdrawing from an ATM. The method takes a shared bank account which supports provisional `withdraw()` and provisional `balance()`, with contingencies `Overdraft` and `UpdatedBalance` respectively. The withdrawal is allowed to proceed provisionally if the chance for overdraft is low; if the chance of overdraft is high then it instead chooses to synchronously `commit` the withdrawal.

3.7.3 *Information Flow in Branches*

The `peek` operation introduces the ability to have differently-consistent values interacting within the same program context. To ensure the resulting *mixed consistency* cannot lead to violations of strongly-consistent guarantees, we turn once again to MixT's type system from chapter 1.

Gallifrey tracks provisionality using an adaptation of MixT's information flow type system.

In an *information flow* type system, values are associated with labels drawn from a lattice. Unlike in MixT, Gallifrey does not reason directly about consistency, choosing instead to reason about specific consistency violations visible through provisional methods. Thus Gallifrey's lattice does not contain consistency models; instead, it contains elements that are sets of provisional methods, ordered by subset inclusion. Each value is labeled with the set of provisional methods which have influenced it. Values labeled with the empty set (indicating they have not been influenced by provisional behavior) live at the bottom of the lattice ($\bot$), while values which have been influenced by all possible provisional behavior live at the top ($\top$). To prevent computation from depending on provisional observations, information should be influenced only by information whose label is a subset of that of the influenced information, just like in MixT. Recall that information flow handles both direct influence, like assignment, and indirect influence, like control flow.

Every reference and variable in Gallifrey—including `shared` references and even branch tokens—is associated with one of these provisional labels. Most references and variables will have the $\bot$ label, indicating an absence of provisional behavior. Branch tokens are somewhat special; branches contain computation, and so their labels indicate the set of provisional methods that have been called within them. Similarly, in order to call a provisional method on a shared reference it must be possible to type that reference with an appropriate provisional label—which in turn means it must reside within a branch that can be typed with the appropriate label.

Provisional labels define precisely where the effects of provisional behavior may be visible, enabling the safe use of `peek`. With *`token`*`.peek[`*`ref`*`]`, users can read a value from a branch and use this value outside of the branch's scope. This value's label contains the provisional operations from the branch which have influenced it. For example, a user can use a `peek`ed value to decide whether its branch should be synchronously `commit`ed or asynchronously `pull`ed (Figure 3.9 line 6).

Our information-flow types also delay the point at which contingency callbacks for peeked values must be supplied. This is because contingencies are only necessary when provisional operations have influenced some visible action; in Gallifrey, visible actions can only be influenced by values with the empty provisional label ($\bot$). Thus our information-flow type system will prevent a `peek`ed value from influencing a visible action unless the peeked value is *endorsed*, an operation which downgrades its label so that it can be used in contexts that do not allow influence from provisional operations. It is at this point of endorsement that the user must provide contingency callbacks. For example, a user might `peek` a value from a branch, transform it, and print the result; it is at the point of printing the result that the user must endorse the peek, making it easy to supply callbacks which apologize for the observed effect of the `peek`—the printed value.

3.8 RELATED WORK

Handling conflicts in concurrent operations. A recent trend is to treat conflict handling strategies as part of a shared object's implementation,

as seen in the literature on conflict-free replicated data types [Shapiro, 2017] (CRDTs) and as seen in programming models such as Bloom [Alvaro, Bailis, et al., 2013], Cloud Types [Burckhardt, Fähndrich, et al., 2012], Lasp [Meiklejohn and Van Roy, 2015], and others [Houshmand and Lesani, 2019; Kuper and Newton, 2013; Sivaramakrishnan, Kaki, and Jagannathan, 2015]. Earlier systems—like Bayou [D. B. Terry, Theimer, et al., 1995], Dynamo [DeCandia et al., 2007], and others [Crooks et al., 2016; Schönig, 2015]—often specify conflict handling separately from an object's implementation. But these systems do not ensure that conflict handling is sensible: they leave the job of merging inconsistent state entirely to the user, inviting errors by allowing partial, incorrect, or even inconsistent merge functions. In Gallifrey, restrictions are defined separately from interfaces, and can be defined without access to implementation internals. However, Gallifrey aims to provide stronger guarantees that existing systems which specify conflict-handling separately from implementation by making restrictions part of a shared object's type, allowing unsafe use of the shared object to be rejected at compile time (i.e. when prohibited operations are used, when a merge function is not exhaustive, when the monotonicity of a test can be violated by an allowed operation).

Speculative operations. To provide higher availability in a georeplicated setting, some systems expose speculative operations in their programming model. Correctables [Guerraoui, Pavlovic, and Seredinschi, 2016] provides a mechanism to speculate on preliminary values returned by weakly consistent operations. If the final values returned by strongly consistent operations do not match the preliminary values, then Correctables allows programmers to recompute or discard the effects of the initial speculation. PLANET [Pang et al., 2014] provides callbacks that fire depending on

what stage a transaction is in before a specified timeout, also allowing users to specify a stage when it *speculatively commits* (i.e., will commit "if all goes well", with some explicit probability that all will go well). It also provides callbacks that fire when the final status of the transaction is known, allowing users to execute *apologies* [Helland and Campbell, 2009] when it was speculated to have committed but was ultimately aborted. In a different setting, Concurrent Revisions [Burckhardt, Baldassin, and Leijen, 2010] provides an intuitive programming model for parallel programs by allowing "revisions" to fork off the state of objects and then to join revisions back into their parents by merge functions specified using *revision types*. Gallifrey takes a similar approach, allowing programmers to speculate at the language level within explicit *branches* (Section 3.7.2) that fork off the state of shared objects. Branches can be used without coordination among replicas, in which case Gallifrey requires our own notion of apologies via contingency callbacks; with coordination, branches enjoy strong consistency—no apologies needed.

Coordination avoidance. Work on coordination avoidance in distributed databases has shown that nodes need to coordinate only when they would otherwise execute operations which violate specified invariants [Bailis, Fekete, et al., 2014; Balegas et al., 2015; Magrino et al., 2019; Preguiça et al., 2003; Roy et al., 2015; Whittaker and Hellerstein, 2018]. Gallifrey's *restrictions* (Section 3.3) embody this principle by refining the interface of a shared object such that only specific operations are available at every replica. Restrictions are a *type-safe* mechanism for coordination avoidance, rejecting programs that violate invariants at compile time. In particular, Indigo presents a framework for users to develop replicated objects which allow commutative operations [Balegas et al., 2015]. Indigo allows

the programmer to specify pre- and postconditions, used to statically determine which pairs of operations may conflict. When operations are determined to conflict, Indigo's compiler inserts appropriate code to use *reservations* [Preguiça et al., 2003] in a way that is analogous to Gallifrey's restrictions. We similarly use pre- and postcondition annotations to determine when operations conflict in checking for the exhaustiveness of merge functions. Additionally, we use these annotations to check that the monotonicity of tests are not violated by allowed operations in the restriction. Unlike Gallifrey Indigo does not support orthogonal replication; its analysis is performed on the object interface, while ours is performed on the restrictions.

3.9 FUTURE WORK

We now give a high-level description of open questions, potential challenges, and possible solutions left as challenge problems.

3.9.1 *Extensions to the Language Design*

Bootstrapping. Our language as described to this point works well for objects which require symmetric replication across a potentially unbounded group of nodes. It is mute on the question of bootstrapping: how does a newly-started node initially receive a replicated object to use? For this, we take inspiration from Fabric [Liu, George, et al., 2009] and provide syntax by which a program can name a global variable located on some other Gallifrey node. Concretely, we suggest the syntax

`gal://hostname.tld/TypeName/Restriction/instance_name` to name the global object `instance_name` of type `shared[Restriction] TypeName` located on the machine at `hostname.tld`.

Typestate and reopening branches. Earlier in this paper, we mentioned that Gallifrey programmers can enter and exit in-progress branches. We propose the syntax *`token`*`.open(args...){body}` by which programmers can re-enter a branch, passing it new references to own and giving it a new body to execute. The `body` here has access to all the objects the branch already owns in addition to the ones newly passed in via `open`. To further refine our information-flow type information and to enable the *`token`*`.open` feature, we propose extending Gallifrey with *typestate* on branch tokens. With typestate, linear items can acquire additional labels on their types as the program evolves. Combining information flow with typestate yields a novel variant of *statically tracked*, *flow-sensitive* information flow. For *`token`*`.open`, this means that provisional behavior introduced during `open`'s body does not require a provisional label on the token *before* the point of `open`. Using this we can also extend `abort` and `pull`, allowing programmers to recover (via `peek`) `unique` objects owned by branches even after they have completed.

Actors. Gallifrey's replicated objects are best suited to a setting where all replicas are peers; we cannot comfortably capture concepts like "all nodes may perform some operations and a designated owner node may perform some additional operations". To support explicitly centralized objects, Gallifrey should include a native notion of unreplicated actors [Hewitt, Bishop, and Steiger, 1973]. We have not yet explored how actors fit into the design of Gallifrey.

Subtyping on restrictions. We desire subtyping on restrictions for two reasons. First, we would like to make it easy for users to write parametric code. It should not be an error to pass a more permissive restriction (i.e., more operations allowed) to a function that expects a less permissive restriction. The second is for encapsulation: a programmer may wish to expose a reference to a shared object via a restriction that permits fewer operations on that object, retaining the more permissive restriction for themselves. To implement subtyping, we plan to view restrictions as records of their allowed operations (with contingencies) and use standard width subtyping on records.

Extensions to monotonic tests. Monotonic tests are used with `when` blocks to set up a trigger. Once the condition in the `when` block becomes true, then the body of the block executes. We believe the language of expressions within the `when` block's condition could be enriched. In general, it should be safe to use any function on `tests` as part of a `when`'s condition so long as those functions are monotonic with respect to boolean ordering. We hope to take advantage of recent work by Clancy and Miller [Clancy and Miller, 2017] to statically prove such functions monotonic and thus safe for use in triggers.

Cursors. Our current typing rule for `restrictions` prevents any methods which return an object that may be connected to the receiver. We propose extending the syntax of `restrictions` to allow such objects by applying a *subordinate* restriction to the return result of their receiver-exposing methods. The total restriction over the object consists therefore of the restriction on the object itself, and all restrictions required on the returned result of that object's methods, (and restrictions applied to the results of those methods, etc). The results of these methods would behave as

cursors into the shared object, allowing programmers to continue invoking correctly-restricted operations against a refined interface.

3.9.2 *Implementation Considerations*

We envision Gallifrey as supporting the next generation of wide-area replicated applications. To support heterogeneous applications, and especially to support our proposed branching feature, requires significant enhancements to the current prototype. We enumerate some of these here.

Consistent synchronous branch merges. As mentioned in Section 3.7.2 branches with provisional operations can be synchronously committed without risking provisional conflicts, giving programmers access to the strong and expressive semantics of traditional transactions. We must strive to make this *transactional commit* operation usable. In particular it must be typically fast, for otherwise programmers will be tempted to fall back to asynchronous `pulls`, inviting more provisional behavior than they may truly require. A key challenge introduced by Gallifrey is its tendency toward disconnection; it will be necessary to carry out these commits with high probability even in the presence of intermittent disconnection.

Exposing flexibility to the user. There are many difficult trade-offs and design decisions to be made in Gallifrey's runtime. These trade-offs are necessarily influenced by the particular Gallifrey deployment in question: is the application running across data centers, or across phones? When looking beyond the current Gallifrey prototype, we must strive to expose such choices to the Gallifrey user. Only the user is an expert in their deployment domain.

3.10 CONCLUSION

Our forays into Gallifrey represent a promising pathway towards better concurrent, distributed programming. With *restrictions*, Gallifrey separates *what* can be replicated from *how* it is shared, and provides a statically enforced mechanism for ensuring consistent access to replicated objects. With *branches*, Gallifrey unifies threads, transactions, and replicas into a single intuitive construct. With *contingencies*, Gallifrey provides some sanity to working with weakly consistent state, allowing explicitly scoped violations of isolation and consistency.

Taken together, these features represent a compelling answer to the question of how to write distributed, concurrent, programs with replicated data. With our implementation, we can see that Gallifrey's core features—restrictions, restriction variants, and transitions—already provide a powerful platform for distributed programming.

4

A TYPE SYSTEM FOR ISOLATION

4.1 INTRODUCTION

Chapter 3 presented the design of Gallifrey, a new language for geodistributed programs. For its core correctness guarantees, Gallifrey relies on *restriction safety*: that every interaction with a shared object is made via a `restriction`, which limits an object's interface to only those methods which are safe to call concurrently. These `restrictions` in turn relies on a notion of *isolation*: that everything reachable via a restriction-guarded object is *only* externally reachable via that object.

This pattern—that safe concurrent code is achieved through isolation—is not unique to this work. It lies at the core of Rust, where the Rustbelt project takes a "fiction of separation" approach to use separation (or isolation) as a proxy for thread safety. It underpins the object-oriented race-free abstractions work popular during the turn of the millennium. Under a different guise, it's even present in languages with safe manual memory management, where capability-based or region-based type systems use isolation to ensure programs only access *reserved* memory—leveraging the intuition that "unreserved" memory may in fact belong to some other context, with very different ideas of how that memory should be used.

The core idea that all of these systems share—and that Gallifrey leverages—is that there is some set of *reserved* objects which can be safely

accessed in a given context, and that access to any other object might lead to undesirable behavior—for example a destructive race between two threads or access to uninitialized memory. These systems provide *reservation safety*: a correct program will never attempt to access memory outside of its reservation. Each of these systems then proposes an intricate static approach to prove reservation safety, preventing the programmer from relying on costly runtime mechanisms or best practices.

But these systems involve significant complexity—much of which comes from managing object graphs. It is tempting to say that whenever an object is reserved, so too are all objects reachable from it; indeed, several existing solutions do just this [Aldrich, Kostadinov, and Chambers, 2002; Almeida, 1997; Gordon et al., 2012; Hogg, 1991; Minsky, 1996; Naden et al., 2012]. But in an object-oriented program the vast majority of objects will be connected in some way; to align reservations (the set of safely-accessible objects) with connected components of the object graph is therefore too coarse an approximation.

The challenge is to allow references that link objects in different reservations, without risking reservation safety. There is however no consensus on how to accomplish this. Existing approaches have ranged from explicitly tracking all references [Protzenko et al., 2017], to requiring the object graph have some regular tree shape [*Rust Programming Language* 2014], to simply relying on the ability to implicitly null out references which would otherwise point outside a reservation [Aldrich, Kostadinov, and Chambers, 2002]. None of these systems captures exactly what Gallifrey needs. Gallifrey requires a system with the following properties:

- **Minimizes programmer-facing complexity**. We should not need to significantly expand programmer-supplied annotations beyond the keywords already introduced in chapter 3.

- **Maintains object-oriented abstractions**. Our approach needs to work seamlessly with function abstraction, interfaces, inheritance, subtyping, and *crucially* must assume pervasive mutability.

- **Allows arbitrary object graphs**. While many data structures are naturally tree-like, certain useful data structures—such as doubly-linked lists—are not. These must be expressible.

- **Allows fine-grained changes to reservations**. Common actions, such as removing an item from a queue and sending it to a new thread, dynamically change the set of reserved objects. It must be possible to capture this without sacrificing access to swaths of objects in the process.

- **Never implicitly nulls a reference**. Relying on implicit nullability effectively turns every reference into an option type, requiring constant careful reasoning or defensive programming.

To address these challenges, this chapter presents a new type system synthesizing the idea of static *regions* as defined by Tofte and Talpin [Tofte and Talpin, 1994] and the *focus* mechanism from Fähndrich and DeLine's Adoption and Focus [Fähndrich and DeLine, 2002]. A *region* is a statically-known set of objects which have an arbitrary, statically-unknown number of links between them. With regions, each *reservation* (the precise set of safely-accessible objects) is approximated by a *region reservation*: a set of regions, such that an object in any region of a region

reservation must also be a member of the true, dynamic reservation it approximates. Statically, reservations are managed at the level of these regions; as objects are removed from a reservation (perhaps via sending them to some other thread) or added to a reservation (perhaps by receiving them from some other thread), the corresponding regions are removed or added to the region reservation. This eliminates the need to track the precise relationship of every object in an object graph; by manipulating reservations at the level of regions, a reservation can gain or lose a set of connected objects *without* needing to statically understand precisely how they are connected.

But we also need to describe connections between objects which cross regions. By default, object fields may only contain references to other objects within the same region; a special `isolated` keyword refers to fields containing references whose target may live in any region. The targets of `isolated` fields must be precisely statically tracked; for this we adopt Fähndrich and DeLine's "Adoption and Focus" mechanism.

We accomplish this with surprisingly little user-facing complexity. Users need only decide which parts of their object graph should be separable from the whole, and annotate those fields with the `isolated` keyword. From there, users also must annotate function parameters to indicate if they `consume` or `preserve` their arguments. Everything else is inferrable.

This relative simplicity in user-facing code is enabled by an observation: if a programmer intends an object subgraph to be easily separable from its parent, then there should not be many references from the parent into that object subgraph, and all such references should be easy to find. In fact, in the most common case there will be exactly one such reference, contained in a single `isolated` field. This pattern has emerged for safety; once an

object has been deleted, sent to a new thread, or had its lock released, any further attempts to access it would be unsafe. The more references exist to that object, the easier it is to use one of them accidentally. Following this pattern, we introduce the idea of a *simple* object subgraph as one in which all `isolated` references point to otherwise-unreachable regions. This definition of simplicity captures the idea that most regions will be reachable by only a single `isolated` link. By making simplicity default, we need only explicitly track deviations from this simplicity—as may arise during object construction or in the process of swapping `isolated` fields. These deviations can be inferred so long as they do not cross abstraction boundaries.

4.2 OVERVIEW

4.2.1 *An Intuition for Reservations*

A *reservation* is the set of objects that a given context may safely access. For example, in a concurrent language the reservation for each thread may consist of the objects which only that thread can name, plus any objects guarded by a lock that thread currently holds. Certain system events can cause a change in the reservation. For example, sending a message to a new thread would *remove* the objects that comprise the message from the sending thread's reservation, and *add* them to the receiving thread's reservation. For the purposes of this chapter, we do not address how reservations are generated. In our formal treatment, the reservation—

which is just a set of locations—is an explicit member of the configurations over which our small-step semantics (4.7.3) are defined.

Using these reservations, we statically guarantee *reservation safety*: that no step of evaluation will attempt to access a location outside of its reservation.

4.2.2 *An Intuition for Regions*

To provide a static guarantee of reservation safety requires a type system to statically and conservatively approximate reservations. Doing this precisely—with static knowledge of every reference to every individual object—swiftly becomes undecidable for realistic languages. Instead, the type system does this at the level of *regions*. Regions group objects which may be connected without needing precise static information about *how* those objects are connected. Sets of regions approximate a reservation: all objects within this set of regions must also lie within the reservation it approximates. Events which change a reservation—for example sending objects across thread boundaries—do so at the granularity of regions. This frees the type system from needing to statically track every reference directly; it must now only statically track those references which cross regions.

Tracking cross-region references statically requires a trade-off between expressiveness at the cost of a high annotation burden, or low programmer-facing complexity at the cost of restrictions on the shape of the object graph. We choose a middle road: we require minimal programmer annotations

in the default case, with additional annotations required only when the object graph deviates from a *simple* shape.

In particular, we first require that all potential cross-region references are annotated with a special `isolated` keyword. References marked `isolated` may refer to any object in any region; references *not* marked `isolated` may only refer to objects in their own regions.

Next, we define a *simplicity* property. We define a simple *region* in terms of the *region graph*, an abstraction of the object graph. Nodes in the region graph are regions, and a directed edge exists between any two regions if an object in one region contains a reference to an object in the other. Using this graph, we say a region is *simple* if the graph of regions reachable from it forms a tree. We similarly define a *simple* `isolated` reference as one whose target is a simple region, and a simple object as one all of whose `isolated` fields are simple. As a program executes, certain simple references, regions, and objects may become non-simple, and certain non-simple references, regions, and objects may become simple. This is desirable; normal patterns—such as swapping two `isolated` fields—pass through temporary non-simple states, only to return to simplicity moments later. When swapping references, there will be a point after the first field has been replaced (but before the second) when both first and second point to the same region, a temporary violation of simplicity.

At function boundaries, we require that arguments' and returned results' regions satisfy this simplicity property. This invariant of *simple* regions, combined with the program text of the function body, provides enough information to infer the targets of cross-region references during type-checking, regardless of their simplicity.

Formally, a region is a set of objects wherein all references from an object inside the region to some object outside of the region are marked `isolated`. No two regions may intersect.

4.2.3 *A Running Example*

To concretize the ideas of regions and reservations we introduce the example of a doubly-linked list, backed by the Java class `LinkedList<T>`:

Example 4.2.1. A doubly-linked list.

```
class LinkedList<T>{
  static class ListNode {
    ListNode next;
    ListNode prev;
    T payload;
  }
  ListNode head;
};
```

Instances of the outer `LinkedList<T>` class contain a single `head` field. This field points to an instance of the inner `ListNode` class, which contains references to the next node in the list, the previous node in the list, and a `payload` element stored in this list node. Figure 4.1 illustrates an instance of this class.

To use this linked list without violating reservation safety, it must first be divided into regions. As regions are the granularity at which reservations gain or lose objects, this is effectively the same question as determining which parts of this linked list should be detachable. One must balance

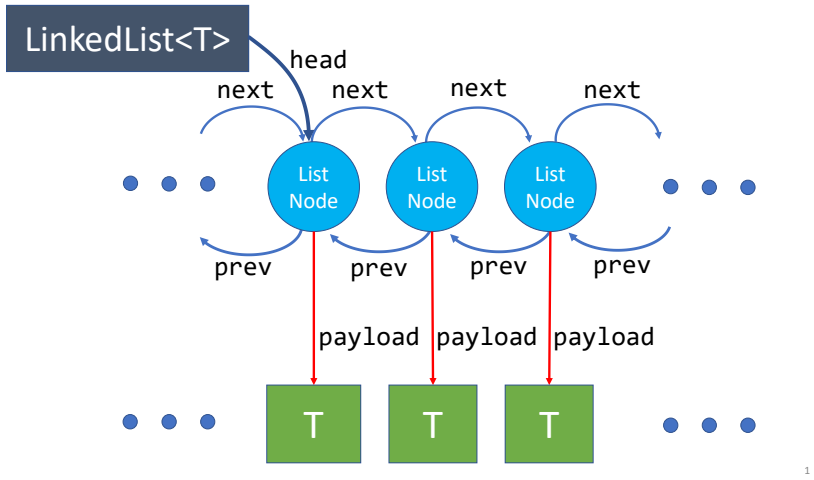


Figure 4.1: An instance of a doubly-linked list. The rectangle labeled "LinkedList" is an instance of the outer LinkedList class, while the circles labeled "ListNode" are instances of the inner ListNode class. The arrows represent references connecting the objects; the dashed arrows are meant to indicate that an arbitrary number of ListNode instances may precede or follow the ones illustrated here.

this against the need to ensure such references are *simple* at function boundaries: the region graph must form a forest connected via `isolated` references.

The `payload` references of the list nodes nicely satisfy both criteria. It is natural to remove items from a list and send them to a new thread; to support that functionality, the regions in which each `payload` resides must all be distinct, and none may contain the list itself. Checking our simplicity criteria, if each `payload` lives in a separate region and the remaining objects live in a single `list` region, then the regions clearly form a tree. Figure 4.2 illustrates this region assignment; here, the red rounded rectangles indicate regions.

An even more precise region assignment is also possible. One should always seek to find the most-precise region assignment which preserves

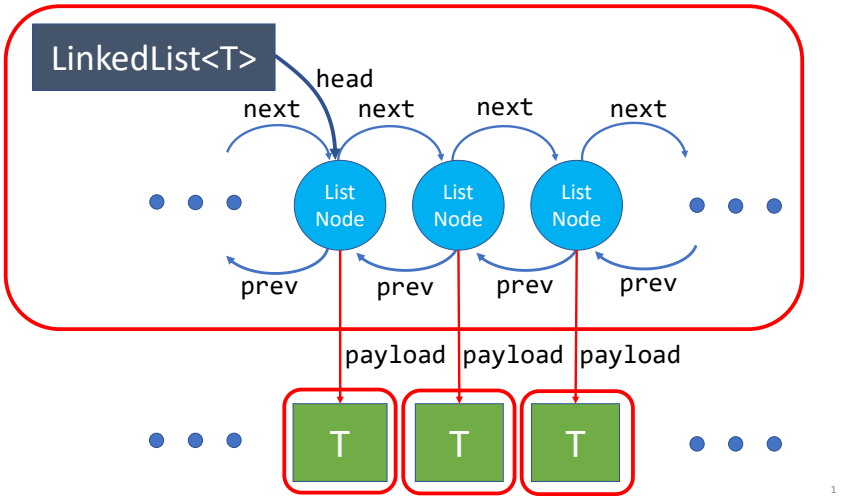


Figure 4.2: A revision of 4.1 in which the rounded rectangles indicate regions.

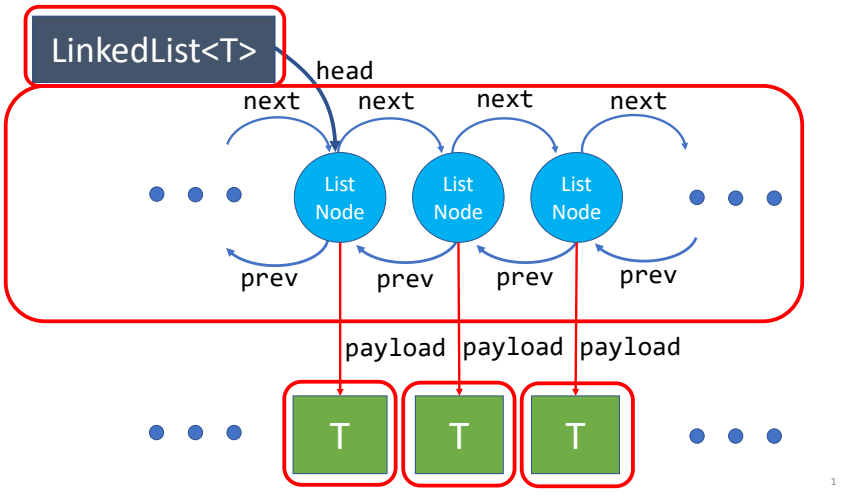


Figure 4.3: A revision of 4.2 in which we have separated the LinkedList into a separate region from the ListNodes.

simplicity; the more precise the region assignment, the more accurately it will reflect the true reservation as regions are removed and added. The primary drawback of having more regions is the need to ensure they are simple at function boundaries; a winning strategy for determining regions is thus to look for the most precise region assignment which still forms a tree. Using this heuristic reveals that the governing `LinkedList` instance contains a reference only to a `ListNode`. As there should never be any path from that `ListNode` *back* to the `LinkedList` instance, the `LinkedList` instance can have its own region without violating simplicity. This final region assignment is illustrated in 4.3.

Syntactically, this region assignment is described by applying the `isolated` keyword to both the `head` field of `LinkedList<T>`, and the `payload` field of `ListNode`, as shown below:

Example 4.2.2. A doubly-linked list.

```
class LinkedList<T>{
  static class ListNode {
    ListNode next;
    ListNode prev;
    isolated T payload;
  }
  isolated ListNode head;
};
```

This modest amount of annotation is all that is required of the user, nicely satisfying the goal of reducing user-facing complexity.

4.3 TRACKING REGIONS AND RESERVATIONS STATICALLY

The remainder of this chapter builds out a type system for a language with regions iteratively. We will start with only sequences, variable declaration, field reference, and field assignment, assuming that complex objects—such

$$
\begin{aligned}
e &::= e; e \mid x \mid x = e \mid e.f \mid e.f = e \\
\tau &::= Cls \mid \ldots \\
\mathit{fields}(Cls) &::= \{f : q_r\ \tau, \ldots\} \\
q_r &::= \texttt{isolated} \mid \cdot
\end{aligned}
$$

Figure 4.4: A syntax of sequences, structures, and assignment. The f metavariable ranges over field names; the x metavariable ranges over variable names; the Cls metavariable ranges over class names. Our types are just class names; to inspect these types, one can use the *fields* function to project out the set of fields contained in the class. These fields are typed, and each may be annotated with the `isolated` keyword.

as the running doubly-linked list example—are already bound in our initial context. We will then extend this language to include functions and the familiar language constructs of IMP. After this point, we will introduce a dynamic semantics for our language and demonstrate that our type system proves reservation safety for this language. We then introduce an extension to the core language, discuss related work, and conclude.

4.3.1 *A Language and Typing Contexts for Structures*

The core of our type system is its treatment of structures. We thus start with a tiny language containing the bare minimum needed to discuss structures.

The syntax for this language can be found in figure 4.4. This syntax describes classes which can be referred to by name, and whose fields may listed via the *fields* function. Each field may be qualified by the `isolated` keyword. The syntax is otherwise completely standard. The semantics is that of java; field access returns a reference, and assignment overwrites the

value of the reference on the left-hand side with the value of the reference on the right-hand side. We choose Java semantics here to better match Gallifrey, which is implemented as an extension to Java. The formal small-step semantics of this language can be found as part of the full language in section 4.7.3.

This language needs a type system whose static typing contexts are capable of tracking the destination regions of all `isolated` references. This is no easy task; as illustrated by the dashed lines in figure 4.3, there are in fact arbitrarily many regions contained even in the running linked list example. To capture this in a finite context, we turn to the idea of *simplicity*. As discussed previously, a *simple* `isolated` reference dominates its object graph: all paths from roots (e.g. references on the stack) to any object reachable via a simple `isolated` reference must transit that isolated reference. Put differently, if a *simple* `isolated` reference were to be set to null, then the entire object graph reachable from it (before it was set to null) is now unreachable. Further, all objects reachable from a simple `isolated` reference must be in the current reservation. This provides a default structure to work with, freeing the static contexts to only track *exceptions* to simplicity.

Building on this idea, we define two static typing contexts. First is Γ, which binds variables. All variables in this language contain references; as such, Γ associates variables with both a type τ and a region ℓ in which the object pointed to by the variable lives: $x : \ell\ \tau \in \Gamma$. Next comes $\mathcal{H}$, which describes our *heap context*. This context is responsible both for tracking the current reservation, *and* for tracking the source and destination of every `isolated` reference. $\mathcal{H}$ is formalized as a set of regions; all regions in $\mathcal{H}$,

$$
\begin{aligned}
\Gamma &::= x : \ell\ \tau, \Gamma \mid \cdot \\
\mathcal{H} &::= \ell\langle\rangle, \mathcal{H} \mid \ell\langle X\rangle, \mathcal{H} \mid \cdot \\
X &::= x[F], X \mid \cdot \\
F &::= f \mapsto \ell, F \mid \cdot
\end{aligned}
$$

Figure 4.5: The definition of typing contexts Γ and $\mathcal{H}$. Here ℓ names regions. As before, f names fields and x names variables.

$$
\frac{\forall x : \ell\ \tau \in \Gamma,\ \ell'\langle x[\ldots], \ldots\rangle \in \mathcal{H} \colon \ell = \ell'}{\vdash \Gamma; \mathcal{H}}
$$

$\Gamma;\mathcal{H}$ WELL-FORMED

Figure 4.6: A well-formedness condition on Γ and $\mathcal{H}$, requiring any variables associated with some region in Γ are associated with that same region in $\mathcal{H}$.

as well as all regions reachable from *simple* `isolated` fields of objects in those regions, are known to be in our current reservation. $\ell\langle\ldots\rangle \in \mathcal{H}$.

The regions in $\mathcal{H}$ have additional structure beyond just a name. Because not every object in a region is guaranteed to be simple at all times, the regions in $\mathcal{H}$ need a way to describe any objects that violate simplicity. This is done by explicitly listing such objects within the region. Each object is referenced via some variable name from Γ, and is associated with a set of fields which reference some other named region. $\ell\langle x[f \mapsto \ell', \ldots], \ldots\rangle \in \mathcal{H}$. The syntactic definitions of both Γ and $\mathcal{H}$ can be found in figure 4.5. While we use the syntax of lists in defining these structures, they are actually unordered; **these environments are sets**. Thus if a rule appears to match on the first or last element of some environment, it should instead be read as matching on any element contained in the environment.

Figure 4.6 states a well-formedness condition for Γ and $\mathcal{H}$: we say that a pair of environments Γ and $\mathcal{H}$ are well-formed when if a variable explicitly

tracked in $\mathcal{H}$ is bound in Γ, then that variable is tracked in and bound to the same region in both $\mathcal{H}$ and Γ.

4.3.2 *Describing Objects via Static Environments*

This subsection presents some example objects, and shows how the structure of those objects is encoded in Γ and $\mathcal{H}$. To start, recall the linked list from 4.2.2. In all these examples a linked list is bound to the variable x.

In a *simple* state, this linked list—no matter its size— can be captured via the following context:

$$\Gamma = \texttt{x} : \ell\ \texttt{LinkedList<T>};\quad \mathcal{H} = \ell\langle\ \rangle$$

In which x refers to some simple linked list, and ℓ is an arbitrarily-chosen name for the region in which this list lives.

At first glance, this might be surprising; after all, the illustration of this list (figure 4.3) included an unbounded number of regions, yet this static context mentions only one. This is because our linked list, as illustrated in (figure 4.3), is entirely *simple*. Every `isolated` reference in this list refers to some new, otherwise-unreachable region; one needs only to explicitly mention these regions in $\mathcal{H}$ when some other reference can reach them. Nevertheless, $\mathcal{H}$ is also able to explicitly include some of them. The following context also captures the linked list:

$$\Gamma = \texttt{x} : \ell\ \texttt{LinkedList<T>};\quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{head} \mapsto \ell']\rangle, \ell'\langle\ \rangle$$

Here $\mathcal{H}$ explicitly records that the object referred to by x has a field head which references some region ℓ'. As ℓ' contains no explicit contents in $\mathcal{H}$, it must be the case that all objects in ℓ', and all objects other than x in $\mathcal{H}$, are simple.

The primary use of this extended context is not to add detail to simple regions, but rather to express non-simple ones. Given a linked list bound to x, the following assignment results in a non-simple region:

```
y = x.head
```

Here, the isolated reference x.head has been aliased into y, providing a path by which the head of the linked list may be directly referenced—implying that x.head no longer dominates its reachable object graph. Moreover, y and x.head both refer to some object in the same region. This is captured in the following context:

$$\Gamma = \texttt{x} : \ell\ \texttt{LinkedList<T>}, \texttt{y} : \ell'\ \texttt{ListNode};\quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{head} \mapsto \ell'] \rangle, \ell' \langle\ \rangle$$

As $\mathcal{H}$ explicitly lists x with a field head which refers to an object in ℓ', Γ must declare that y is also bound to an object in ℓ'. This extra detail tracks the fact that y and x.head both point to the same region; if x.head leaves the reservation, y is removed as well.

This context can be used to describe more interesting irregular structures as well. For example, consider the following code (again where x is bound to a linked list):

```
y = x.head; z = y.payload; consume z
```

Here `consume` is some special keyword that causes the reservation to drop `z`. After the execution of this fragment, the state of the list can be tracked by this context:

$$\Gamma = \texttt{x} : \ell\ \texttt{LinkedList<T>}, \texttt{y} : \ell'\ \texttt{ListNode}, \texttt{z} : \ell''\ \texttt{ListNode}$$

$$\mathcal{H} = \ell\langle \texttt{x}[\texttt{head} \mapsto \ell'] \rangle, \ell' \langle \texttt{payload} \mapsto \ell'' \rangle$$

Here both `x` and `y.payload` are bound to the region ℓ'', which is omitted from $\mathcal{H}$. This renders `x` and `y.payload` inaccessible.This works well with the invariant on $\mathcal{H}$: all `isolated` fields which are *not* explicitly tracked in $\mathcal{H}$ refer to simple regions contained within the heap reservation (✓), and all regions explicitly listed in $\mathcal{H}$ are also within the heap reservation (✓).

$\mathcal{H}$ is capable of capturing exceptions to simplicity beyond the validity of certain `isolated` references; explicitly-tracked `isolated` fields are not constrained to a tree shape, and thus can capture an arbitrary object graph—so long as one is willing to explicitly enumerate it. Here is a silly example in which the `head` node of the linked list actually lives in the same region as the `LinkedList` instance, despite the presence of the `isolated` keyword on `head`:

$$\Gamma = \texttt{x} : \ell\ \texttt{LinkedList<T>}; \quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{head} \mapsto \ell] \rangle$$

Here `x`, which lives in region ℓ, has some explicitly-tracked `head` field whose target *also* lives in region ℓ.

An important restriction is that $\mathcal{H}$ *only* tracks `isolated` fields; all non-isolated fields are restricted at all times to refer to objects within their region. It is impossible, for example, to capture a `ListNode` whose `next`

reference crosses regions. But from this limitation comes a great deal of freedom; objects in the same region can be arbitrarily linked via non-`isolated` references, without requiring these references explicitly appear in a static typing context.

Intermixing both `isolated` and non-`isolated` references plays both sides of the trade-off inherent in static reference tracking. On one side we can form references between objects in the same region without worrying about the decideability of tracking them, but in exchange must assume the worst case: that *all* objects in a region are connected. On the other side, we have precise knowledge of where references point, and so need make no such connectivity assumption; but the cost of this knowledge is a large footprint in the static context, which would be difficult to infer across abstraction boundaries.

4.3.3 *A Type System for Manipulating Structures*

Building on the definitions of Γ and $\mathcal{H}$, we now introduce a type system for the language fragment in figure 4.4. In presenting this language, we have chosen to break the rules out into a set of core "typing" rules (figure 4.7) and a set of "heap" rules (figure 4.8). We do this for presentation; building an abstraction of heap manipulation syntactically removed from that of the traditional aspects of typechecking allows us to present the essence of our system—the transformation of regions—without the ancillary aspects of typechecking. While the presentation makes it appear as though there is a separate language of heap manipulation, this is not so; every heap rule is referenced directly in the premise of some typing rule.

VARREF

$$\frac{\mathcal{H} \vdash \ell}{\mathcal{H}; \Gamma, x : \ell\ \tau \vdash x : \ell\ \tau \dashv \mathcal{H}; \Gamma}$$

SEQUENCE

$$\frac{\mathcal{H}; \Gamma \vdash e_1 : \ell_1\ \tau_1 \dashv \mathcal{H}'; \Gamma' \qquad \mathcal{H}'; \Gamma' \vdash e_2 : \ell_2\ \tau_2 \dashv \mathcal{H}''; \Gamma''}{\mathcal{H}; \Gamma \vdash e_1; e_2 : \ell_2\ \tau_2 \dashv \mathcal{H}''; \Gamma''}$$

FIELD REFERENCE

$$\frac{\begin{array}{c}\mathcal{H}; \Gamma \vdash e : \ell\ \tau_0 \dashv \mathcal{H}'; \Gamma' \\ f : q_r\ \tau \in \mathit{fields}(\tau_0) \qquad \mathcal{H}' \vdash q_r\ e@\ell.f : \ell' \qquad \mathcal{H}' \vdash \ell'\end{array}}{\mathcal{H}; \Gamma \vdash e.f : \ell'\ \tau \dashv \mathcal{H}'; \Gamma'}$$

FIELD ASSIGNMENT

$$\frac{\begin{array}{c}\mathcal{H}; \Gamma \vdash e_1 : \ell\ \tau_0 \dashv \mathcal{H}'; \Gamma' \qquad f : q_r\ \tau \in \mathit{fields}(\tau_0) \\ \mathcal{H}'; \Gamma' \vdash e_2 : \ell'\ \tau \dashv \mathcal{H}''; \Gamma'' \qquad \vdash q_r\ e_1@\ell.f = \ell' : \mathcal{H}'' \Rightarrow \mathcal{H}'''\end{array}}{\mathcal{H}; \Gamma \vdash e_1.f = e_2 : \ell'\ \tau \dashv \mathcal{H}'''; \Gamma'}$$

ASSIGN-Γ

$$\frac{\mathcal{H}; \Gamma \vdash e : \ell\tau \dashv \mathcal{H}'; \Gamma', x : \ell_i\tau \qquad \mathcal{H}' \vdash x@\ell_i = \ell}{\mathcal{H}; x : \ell_0\tau, \Gamma \vdash x = e : \ell\ \tau \dashv \mathcal{H}'; \Gamma', x : \ell\tau}$$

Figure 4.7: A set of typing rules for the language from figure 4.4, written with reference to rules in figure 4.8.

Typing judgments have the form $\mathcal{H}; \Gamma \vdash e : \ell\ \tau \dashv \mathcal{H}'; \Gamma'$. This judgment means that the result of e is an object of type τ in region ℓ, and that in the course of evaluating e the "input" environments $\mathcal{H}$;Γ have been transformed into the "output" environments $\mathcal{H}'; \Gamma'$. We treat both Γ and $\mathcal{H}$ linearly, using environment-passing style. This reflects the fact that an expression which modifies $\mathcal{H}$ or Γ does not do so within a syntactic scope; all subsequent expressions must be aware of this modification. For example, in evaluating the sequence `x.f = x.g; x.g = x.f` it is likely important that the second assignment is typechecked under an environment which reflects the results of the first assignment.

We now drill down into the typing rules in figure 4.7. Beyond some syntactic oddities, these rules are largely straightforward. The rule for sequence checks its left-hand expression in its "input" context against an "output" context matching the "input" context of its right-hand side. The result of the sequence is the last expression executed within it; thus the type and region of e_2's result is the type and region of the full sequence result.

The rules for field reference, field assignment, and variable assignment are somewhat more interesting. Both rules just check their subexpressions under the relevant environments, threading through their typing contexts (as in SEQUENCE). But both rules also rely on heap judgments.

Heap judgments come in two different forms, corresponding intuitively to judgments which *query* the heap, written $\mathcal{H} \vdash \varphi$, and judgments for *modifications* to the heap, written $\Gamma \vdash cmd : \mathcal{H} \Rightarrow \mathcal{H}'$. While this is written as though φ and *cmd* are selected from some language, we must emphasize again that *they are not*; rather than think of them as referring to statements

ISOLATED-FIELD-REFERENCE
$$\mathcal{H}', \ell\langle x[F, f \mapsto \ell']\rangle \vdash \texttt{isolated}\ x.f@\ell : \ell'$$

ISOLATED-FIELD-ASSIGNMENT
$$\vdash \texttt{isolated}\ x.f@\ell = \ell' : \mathcal{H}, \ell\langle x[F, f \mapsto \ell'']\rangle \Rightarrow \mathcal{H}, \ell\langle x[F, f \mapsto \ell']\rangle$$

NON-ISOLATED FIELDS
$$\mathcal{H} \vdash \cdot\ e.f@\ell : \ell$$

NON-ISOLATED ASSIGNMENT
$$\vdash \cdot\ e.f@\ell = \ell : \mathcal{H} \Rightarrow \mathcal{H}$$

REGION-VALID
$$\mathcal{H}, \ell\langle X\rangle \vdash \ell$$

UNTRACKED
$$\frac{x \notin \{x_1, \ldots, x_n\}}{\mathcal{H}, \ell\langle x_1[F_1], \ldots, x_n[F_n]\rangle \vdash x@\ell\ \texttt{untracked}}$$

ASSIGNMENT-VALID
$$\frac{\mathcal{H} \vdash x@\ell\ \texttt{untracked}}{\mathcal{H} \vdash x@\ell = \ell'}$$

Figure 4.8: A set of "heap" rules for the type system in figure 4.7.

in a language, think of them as indexing different otherwise-syntactically-identical judgments.

We now pause our exploration of the typing rules to review the heap rules in figure 4.8. The first two rules, ISOLATED-FIELD-REFERENCE and ISOLATED-FIELD-ASSIGNMENT, correspond to assignment and field reference for `isolated` fields. These rules directly use the structure of $\mathcal{H}$ presented in the previous section. To look up the region referenced by the `isolated` field f of some object x in region ℓ, ℓ must be is explicitly tracking some x with field f in $\mathcal{H}$;, the region referenced by f is explicitly listed therein. To assign to some field f of some object x in region ℓ, f must similarly be explicitly tracked; its mapping is then updated from some previous ℓ'' to our new target ℓ'.

The next two rules handle field reference and assignment for *non*-isolated fields. These rules are less complex. As it is a requirement that all non-isolated fields refer to objects in the same region, $\mathcal{H}$ has no role; rather, any

non-isolated field reference from an object in region ℓ returns an object in region ℓ, and assignment is allowed between any two objects in the same region.

Finally, we come to REGION-VALID, UNTRACKED, and ASSIGNMENT-VALID. The REGION-VALID rule checks if ℓ is the name of a region within $\mathcal{H}$. The ASSIGNMENT-VALID rule is slightly more complex; because variable names refer directly to objects within $\mathcal{H}$, we cannot allow assignments to tracked variables. The ASSIGNMENT-VALID rule therefore checks if the variable is tracked by reference to the UNTRACKED rule.

We can now return to figure 4.7 and discuss how the field reference and assignment typing rules use the corresponding heap rules. In FIELD REFERENCE, the expression's result has a field f with the possible `isolated` keyword captured in q_r; the premise includes a field-reference heap judgment with this information, determining first that f refers to an object in some region ℓ', and second that this region is actually within the reservation and thus may be accessed. In FIELD ASSIGNMENT, we similarly determine that the left-hand expression's result lives in region ℓ and has a field f with the possible `isolated` keyword captured in q_r, and that the right-hand expression lives in some region ℓ'. We then use the heap rule for assignment to determine whether this assignment is admissible and what effect it will have on the heap, if any. Variable assignment proceeds similarly, with two important exceptions. First, we do not care if the region labels of the left- and right-hand sides match; as Γ is an affine environment, we can just change the target's label in the output context. Second, this assignment cannot change the heap context; instead, any assignment which would have required a change to the heap context is prevented by the heap rule in the premise.

It may occur to the reader at this point that the typing rules, as presented, require contexts which already reflect the presence of variables in Γ and the precise destinations of `isolated` fields in $\mathcal{H}$; the rules for introducing variables in Γ and determining what to track in $\mathcal{H}$ will be presented later.

4.3.4 *Typechecking Examples*

With this fragment of our system in hand, we can now typecheck some single-line examples. In each example, the typing context comes before each line.

Example 4.3.1. Checking a non-isolated assignment.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode};\ \ \mathcal{H} = \ell\langle\ \rangle$
`x.next = x.next.next` //*uses non-isolated field reference and assignment*
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode};\ \ \mathcal{H} = \ell\langle\ \rangle$

This example shows that checking a non-isolated field assignment does not change the typing context at all; the typing context is only used to ensure that both sides of the assignment live in the same region, and that this region is within the reservation.

Example 4.3.2. Checking an isolated assignment.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ \texttt{ListNode};$
$\mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell_x]\rangle, \ell'\langle \texttt{y}[\texttt{payload} \mapsto \ell_y]\rangle, \ell_x\langle\ \rangle, \ell_y\langle\ \rangle$
`y.payload = x.payload` //*non-isolated field access and isolated assignment*
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ \texttt{ListNode};$

$\mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell_x]\rangle, \ell'\langle \texttt{y}[\texttt{payload} \mapsto \ell_x]\rangle, \ell_x\langle\ \rangle, \ell_y\langle\ \rangle$

As this example illustrates, assignments between isolated references in different regions are allowed; as a result, the heap context is updated to track that `y.payload` now points to ℓ_x, the same region as `x.payload`. This

does, however, mean that the heap is no longer *simple*, as two different `isolated` references now refer to the same region. This is a cost, and it will come due when we introduce functions.

Assignments to `isolated` references need not be from other isolated references, as shown in the next example.

Example 4.3.3. An isolated assignment from a variable.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ \texttt{T};$
$\mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell_x]\rangle, \ell'\langle\rangle, \ell_x\langle\ \rangle$
`x.payload = y` //*uses variable reference and isolated assignment*
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ \texttt{T};$
$\mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\rangle, \ell_x\langle\ \rangle$

Here the `isolated payload` is being set to the value of `y`, which is some existing payload in a different region. This assignment is allowed, and modifies the heap to have `payload` refer to the same region as `y`. It also would have been possible to write this example even if `y` were in the same region as `x` or `x.payload`; the result may not have been simple, but it would be allowed so long as the relevant `isolated` references are explicitly tracked in a region in $\mathcal{H}$.

There are also several more unusual positive examples:

Example 4.3.4. A self-assignment, possible with a `LinkedList<ListNode>`.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode};\quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell_x]\rangle$
`x.payload = x.next` //*non-isolated field access and isolated assignment*
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode};\quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell]\rangle$

This example demonstrates two things. First, `x.payload` is an inaccessible object at the beginning of this snippet; the left-hand side of an assignment need not refer to a valid object, as the assignment expression will overwrite this reference in any case. Second, this example assigns the object at a non-`isolated` field into and `isolated` field. This is no different

than any other assignment to an isolated field; we just update the tracking in $\mathcal{H}$ to point to the correct region. As before, this new heap is not simple. One could also write the reverse of this example:

Example 4.3.5. A less sensible self-assignment, possible with a `LinkedList<ListNode>`.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode};\quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell]\rangle$
`x.next = x.payload` //*isolated field access and non-isolated assignment*
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode};\quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell]\rangle$

Here an `isolated` reference is assigned into a non-`isolated` field of the same object. This is a violation of simplicity, and is only legal here because `x` *already* was non-simple before the assignment, as it lived in the same region to which one of its `isolated` fields pointed.

4.3.5 *Limitations: the Puzzle of Aliasing*

Of course, not all possible programs will typecheck; it is possible to turn every positive example from the previous subsection into a negative example by invalidating one of the context assumptions on which it relies. But there are actually interesting limits with this system beyond straightforward region or variable mismatches. Consider for example the following.

Example 4.3.6. An invalid assignment.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode};\quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\ \rangle$
`x.payload = x.next.payload` //*cannot reference x.next.payload*
`ERROR`.

This example attempts to assign `x.next.payload` into `x.payload`. This will not typecheck; the rule for `isolated` field reference, which we will

need to apply to read `x.next.payload`, requires that the *exact expression* under consideration—here `x.next`—already be tracked in $\mathcal{H}$.

It may in fact be *impossible* to build an $\mathcal{H}$ in which both `x` and `x.next` are tracked. This is due to aliasing. There is nothing in the type or simplicity of `x` to declare that `x` and `x.next` must refer to separate objects. Conservatively, they could be one in the same; each region could in fact contain a single object, and *every* expression whose result lives in this region refers to the same object. In such a situation, it would be an error to include both `x` and `x.next` in the same region in $\mathcal{H}$; as each variable tracked in a region in $\mathcal{H}$ must refer to a *distinct* object.

In point of fact, without knowledge of aliasing we will *never* be able to explicitly track more than one object per region. Without some sort of dynamic check or the integration of an alias analysis oracle, the rules presented here will only be able to track a single object per region.

4.4 VIRTUAL COMMANDS

The rules presented in section 4.3 show how the Γ and $\mathcal{H}$ contexts track modifications to the heap: they require that every use of an `isolated` field is made in a context where $\mathcal{H}$ explicitly tracks the source and destination of that field. But this has not addressed the challenge of tracking `isolated` references so much as moved it; it did not reveal how variables became tracked in $\mathcal{H}$ in the first place. To track individual variables in $\mathcal{H}$, we introduce the idea of *virtual heap commands*.

4.4.1 *Focusing and Exploring*

Consider the following small example program:

Example 4.4.1. Some innocuous assignments.

$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell'''\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell]\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
1: `x1.payload = y;` *//OK: use variable reference and isolated assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell'''\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
2: `x2 = x1.next;` *//OK: non-isolated field reference and variable assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
3: `x2.payload = z;` *//ERROR: `x2.payload` not tracked in $\mathcal{H}$*

Under the rules presented so far, this example will not typecheck, because the variable `x2` is not tracked in $\mathcal{H}$. In fact $\mathcal{H}$ cannot safely track both `x1` and `x2` simultaneously. As `x1` and `x1.next` are different objects, $\mathcal{H}$ cannot include both of them at the start of code fragment. Without knowing if `x1` and `x1.next` are distinct, we aren't sure whether the assignment in line 3 will overwrite the assignment in line 1; at the end of this fragment, it's possible that `x1.payload` and `x2.payload` point to the same or different regions. We run the risk of confusing the regions pointed to by `x1.payload` and `x2.payload`.

But there is a way to make this example typecheck. In section 4.3.2, we mentioned that objects need only be explicitly tracked in $\mathcal{H}$ if they contained non-simple isolated fields; while one *could* explicitly track any isolated reference they desire in $\mathcal{H}$, the *primary purpose* of doing so is to allow that field to violate simplicity. After line 2, there is no longer a purpose in tracking `x1` in $\mathcal{H}$ at all—`y` is the only thing preventing `x1` from being entirely simple, and `y` is never used again.

What we'd like to do here is explicitly invalidate `y` and remove `x1` from tracking in $\mathcal{H}$ entirely, replacing it with `x2` and permitting the assignment in line 3 to go forward. Effectively, we want to move from this context:

$$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T$$
$$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell'] \rangle, \ell' \langle\ \rangle, \ell'' \langle\ \rangle$$

To this context:

$$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T$$
$$\mathcal{H} = \ell\langle\rangle, \ell'' \langle\ \rangle$$

And finally to this context:

$$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T$$
$$\mathcal{H} = \ell\langle \texttt{x2}[\texttt{payload} \mapsto \ell'''] \rangle, \ell'' \langle\ \rangle, \ell''' \langle\ \rangle$$

The first transformation just removes both the region ℓ' and the tracking on `x1` from the heap context entirely. This satisfies the heap context invariants; in order to remove `x1.payload` from tracking `x1.payload` must refer to a region which was (1) not otherwise reachable and (2) not present in $\mathcal{H}$. Just *removing* its target from $\mathcal{H}$ kills two birds with one stone: any existing references to that region will now become unusable (as they point to a region no longer in $\mathcal{H}$), ensuring that `x1.payload` really is the only remaining valid reference with access to that region. At the cost of rendering `y` unusable, $\mathcal{H}$ no longer tracks this object.

FOCUS-HEAP
$$\Gamma, x : \ell\ \tau \vdash \texttt{focus}\ \ell@x : \mathcal{H}, \ell\langle\rangle \Rightarrow \mathcal{H}, \ell\langle x[\,]\rangle$$

UNFOCUS-HEAP
$$\Gamma \vdash \texttt{unfocus}\ \ell@x : \mathcal{H}, \ell\langle x[], X\rangle \Rightarrow \mathcal{H}, \ell\langle X\rangle$$

EXPLORE-HEAP
$$\frac{F = f_1 \mapsto \ell_1, \ldots, f_n \mapsto \ell_n \qquad f \notin \{f_1, \ldots, f_n\} \qquad \ell' \textit{ fresh}}{\Gamma \vdash \texttt{explore}\ \ell@x.f : \mathcal{H}, \ell\langle x[F], X\rangle \Rightarrow \mathcal{H}', \ell\langle x[F, f \mapsto \ell'], X\rangle, \ell'\langle\,\rangle}$$

RETRACT-HEAP
$$\Gamma \vdash \texttt{retract}\ \ell' : \mathcal{H}, \ell\langle x[F, f \mapsto \ell'], X\rangle, \ell'\langle\rangle \Rightarrow \mathcal{H}, \ell\langle x[F], X\rangle$$

Figure 4.9: Virtual heap commands.

With `x1` no longer tracked in $\mathcal{H}$ we're free to take the next transformation step, moving to the symmetric case where `x2` is in $\mathcal{H}$ instead. Note that none of these steps require reasoning about whether `x1` and `x2` are in fact the same or different objects. By ensuring only one of them is tracked at a time, our type system has enforced that no context will ever simultaneously have access to both `x1.payload` and `x2.payload`. This eliminates the possibility of confusing the `isolated` regions reachable from `x1` and `x2`; that in turn eliminates any possible errors arising from their potentially-aliased state. And all it cost was losing access to `y`.

These intuitions are formalized as a set of *virtual* heap commands found in figure 4.9. In the FOCUS-HEAP rule, a region which has no currently-tracked objects may choose to track any variable bound by Γ. This does not change the dynamic heap or reservation represented by $\mathcal{H}$; rather, instead of having an *implicitly* simple x in ℓ, we now have explicitly tracked that simple x. The UNFOCUS-HEAP rule provides the reverse. Here any clearly-simple x tracked in a region can be removed from $\mathcal{H}$.

It is important to note at this point that FOCUS-HEAP and UNFOCUS-HEAP aren't exact opposites. The UNFOCUS-HEAP rule stops tracking any object with no tracked fields, while the FOCUS-HEAP rule starts tracking an object *only in a region with no other tracked objects*. The reason for this once again comes down to aliasing. It is essential that every tracked entry in ℓ refers to a *distinct* object—and the information available in Γ and $\mathcal{H}$ is not sufficient to determine if any two objects are aliases. As mentioned previously, integration with an alias analysis oracle may be one way to safely focus additional variables.

The next two rules in figure 4.9 discuss tracking fields of an already-tracked object. These leverage the invariant that a field not already explicitly tracked in $\mathcal{H}$ must point to some otherwise-inaccessible region within the current reservation. Leveraging this, EXPLORE-HEAP generate a new name for the region to which a newly-tracked field refers, and add this new, simple region to $\mathcal{H}$. Similarly, RETRACT-HEAP stops tracking a field whose target is a simple region in $\mathcal{H}$, removing the target region from $\mathcal{H}$ in the process. This too is to maintain an invariant: if f is not tracked, then the region it refers to cannot be listed in $\mathcal{H}$.

A natural question at this point would be: why have an "`unfocus`" and "`retract`" instead of associating `focus` and `explore` with a scope? As discussed in section 4.5, it's not always possible to `unfocus` or `retract` at the end of a natural scope; moreover, doing so would eliminate the ability to introduce non-simple functions later.

With these new rules in hand, we can now return to our example and begin to see how it might typecheck:

Example 4.4.2. Some innocuous assignments, with virtual heap commands.

$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell'''\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell]\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
1: `x1.payload = y; //`*OK: use variable reference and isolated assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell'''\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
2: `x2 = x1.next; //`*OK: use non-isolated field reference and variable assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
retract ℓ' `//` *OK, but we lose ℓ', rendering* `y` *inaccessible*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\,]\rangle, \ell''\langle\,\rangle$
unfocus ℓ`@x1`
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle\,\rangle, \ell''\langle\,\rangle$
focus ℓ`@x2`
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x2}[\,]\rangle, \ell''\langle\,\rangle$
explore ℓ`@x2.payload`
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x2}[\texttt{payload} \mapsto \ell''']\rangle, \ell'''\langle\,\rangle, \ell''\langle\,\rangle$
3: `x2.payload = z; //`*OK: use variable reference and isolated assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x2}[\texttt{payload} \mapsto \ell'']\rangle, \ell'''\langle\,\rangle, \ell''\langle\,\rangle$

These virtual heap commands must still be integrated into our typing rules. In practice, the typechecker will be free to use these virtual commands wherever it desires, introducing a proof search component into the process of typechecking our language fragment. But formalizing this search process directly will complicate efforts to prove our system correct; instead, we choose to directly reflect these heap commands in the surface syntax of the language, as in figure 4.10. One important detail of our choice here is that we have *not* directly exposed the names of regions in the surface syntax; this will be important later, as we will rely on the fact that these names are chosen arbitrarily.

$$e ::= \ldots \mid \texttt{focus}\ x \mid \texttt{explore}\ x.f \mid \texttt{retract}\ e \mid \texttt{unfocus}\ x$$
$$cmd ::= \texttt{focus}\ \ell@x \mid \texttt{explore}\ \ell@x.f \mid \texttt{retract}\ \ell \mid \texttt{unfocus}\ \ell@x$$

FOCUS
$$\frac{\mathcal{H};\Gamma \vdash x : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{focus}\ \ell@x \dashv H' \Rightarrow \mathcal{H}''}{\mathcal{H};\Gamma \vdash \texttt{focus}\ x : \ell\ \tau \dashv \mathcal{H}'';\Gamma'}$$

EXPLORE
$$\frac{\mathcal{H};\Gamma \vdash x : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{explore}\ \ell@x.f : \mathcal{H}' \Rightarrow \mathcal{H}'' \qquad \mathcal{H}'';\Gamma' \vdash x.f : \ell'\ \tau \dashv \mathcal{H}''';\Gamma''}{\mathcal{H};\Gamma \vdash \texttt{explore}\ x.f : \ell' \dashv \mathcal{H}''';\Gamma''}$$

RETRACT
$$\frac{\mathcal{H};\Gamma \vdash e : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{retract}\ \ell : \mathcal{H}' \Rightarrow \mathcal{H}''}{\mathcal{H};\Gamma \vdash \texttt{retract}\ e : \bot\ \texttt{void} \dashv \mathcal{H}'';\Gamma'}$$

UNFOCUS
$$\frac{\mathcal{H};\Gamma \vdash x : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{unfocus}\ \ell@x : \mathcal{H}' \Rightarrow \mathcal{H}''}{\mathcal{H};\Gamma \vdash \texttt{unfocus}\ x : \ell\ \tau \dashv \mathcal{H}'';\Gamma'}$$

Figure 4.10: Syntax and typing rules for tracking.

Finally we have built up the mechanism required to type the example properly.

Example 4.4.3. Some innocuous assignments, with surface-syntax heap commands.

$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell'''\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell]\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
1: `x1.payload = y;` *//OK: use variable reference and isolated assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell'''\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
2: `x2 = x1.next;` *//OK: non-isolated field access and variable assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$
`retract y //` *OK, but we lose ℓ', rendering* `y` *inaccessible*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x1}[\,]\rangle, \ell''\langle\,\rangle$
`unfocus x1`
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle\,\rangle, \ell''\langle\,\rangle$
`focus x2`
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x2}[\,]\rangle, \ell''\langle\,\rangle$
`explore x2.payload`
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x2}[\texttt{payload} \mapsto \ell''']\rangle, \ell'''\langle\,\rangle, \ell''\langle\,\rangle$
3: `x2.payload = z;` *//OK: use variable reference and isolated assignment*
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ T, \texttt{z} : \ell''\ T;$
$\mathcal{H} = \ell\langle \texttt{x2}[\texttt{payload} \mapsto \ell'']\rangle, \ell'''\langle\,\rangle, \ell''\langle\,\rangle$

4.4.2 *Attaching Regions*

The `focus`, `explore`, `retract`, and `unfocus` virtual commands cause the static contexts to gain (and lose) information about specific objects within a region. To these we add an additional virtual command: `attach`. The `attach` command allows the heap context to "forget" that two regions were distinct, renaming all references to one into references to the other.

$$e ::= \ldots \mid \texttt{attach}\ e\ \texttt{to}\ e$$
$$cmd ::= \ldots \mid \texttt{attach}\ \ell\ \texttt{to}\ \ell$$

ATTACH-HEAP
$$\frac{\mathcal{H}' = \mathcal{H}[\ell_1 \mapsto \ell_2] \qquad X_1' = X_1[\ell_1 \mapsto \ell_2] \qquad X_2' = X_2[\ell_1 \mapsto \ell_2]}{\vdash \texttt{attach}\ \ell_1\ \texttt{to}\ \ell_2 : \mathcal{H}, \ell_1\langle X_1\rangle, \ell_2\langle X_2\rangle \Rightarrow \mathcal{H}', \ell_2\langle X_1', X_2'\rangle}$$

ATTACH-HEAP-NOOP
$$\Gamma \vdash \texttt{attach}\ \ell\ \texttt{to}\ \ell : \mathcal{H} \Rightarrow \mathcal{H}$$

ATTACH
$$\frac{\begin{array}{c}\mathcal{H};\Gamma \vdash e_1 : \ell_1\ \tau_1 \dashv \mathcal{H}';\Gamma' \\ \mathcal{H}';\Gamma' \vdash e_2 : \ell_2\ \tau_2 \dashv \mathcal{H}'';\Gamma'' \qquad \Gamma'' \vdash \texttt{attach}\ \ell_1\ \texttt{to}\ \ell_2 : \mathcal{H}'' \Rightarrow \mathcal{H}'''\end{array}}{\mathcal{H};\Gamma \vdash \texttt{attach}\ e_1\ \texttt{to}\ e_2 : \ell_2\ \tau_1 \dashv \mathcal{H}''';\Gamma''[\ell_1 \mapsto \ell_2]}$$

Figure 4.11: `attach`, for unifying regions. The $[\cdot \mapsto \cdot]$ syntax in this rule indicates a variable renaming, and should not be confused for a tracked field.

Unlike focus and explore, `attach` has no inverse; there is no statically-safe way to split a region into two without extra dynamic information.

To see how `attach` might be useful, consider the following example.

Example 4.4.4. Attempting non-`isolated` cross-region assignment.

$\Gamma =$ `x1` $: \ell$ `ListNode`, `x2` $: \ell'$ `ListNode` $\quad \mathcal{H} = \ell\langle\,\rangle, \ell'\langle\,\rangle, \ell''\langle\,\rangle$

```
x1.next = x2; //ERROR: non-isolated "x1.next" cannot cross regions.
```

This example attempts to set the next list node after `x1` to be `x2`. But this cannot be done: `x1.next` can only form references to the same region as `x1`, while `x2` lives in a different region. To perform this assignment requires "forgetting" that `x1` and `x2` are in different regions—much as example 4.4.3 required "forgetting" that `y` and `x1.payload` lived in the same region.

This can be achieved by inserting an `attach` using the rules in figure 4.11. As before, we have chosen to split this out into a "typing" rule and a "heap" rule. But unlike with focus and explore, the typing rule here

does not serve only to expose the heap rule to the surface syntax; because `attach` effectively relabels one region into another, our typing rule for `attach` is able to perform that relabeling within Γ. This makes `attach`, in a way, less costly to use than `retract`; the attach itself will not render any variables (or fields) inaccessible. It will however complicate future attempts at focus; once two regions are attached, the type contexts forget the fact that objects formerly in one or the other cannot be aliases of each other.

The rules of `attach` in figure 4.11 deserve a more careful look. The heap rule ATTACH-HEAP takes two regions ℓ_1 and ℓ_2 in which both ℓ_1 and ℓ_2 have an arbitrary number of focused objects, and replaces them with a single unified region ℓ_2 containing the focused objects of both original regions. Additionally, it renames all instances of ℓ_1 to ℓ_2 anywhere they occur. Our typing rule first extracts the regions from its two subexpressions, and then uses the heap rule to unify those regions. It continues the process of renaming started by the heap rule, replacing ℓ_1 with ℓ_2 wherever it occurs in Γ.

This new `attach` construct fixes the example:

Example 4.4.5. Attempting non-`isolated` cross-region assignment.

$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell'\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle\,\rangle, \ell'\langle\,\rangle$
```
attach x2 to x1
```
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle\,\rangle$
```
x1.next = x2; //OK: variable reference and non-isolated field assignment
```
$\Gamma = \texttt{x1} : \ell\ \texttt{ListNode}, \texttt{x2} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle\,\rangle$

Where with focus and explore aliasing proved to be our Achilles's Heel, with `attach` our trouble is reachability. Objects within a single region may by connected via non-`isolated` fields; objects in separate regions may be connected only via `isolated` fields. Attaching two regions together is

therefore always safe; but splitting a region in two is much more difficult. To take a region and split it in two would require knowing that there are only `isolated` references which cross the new divide. But we do not track non-`isolated` references directly; indeed, that's the entire point of having regions in the first place. So we cannot easily account for non-`isolated` references statically. A hybrid dynamic and static approach which allows splitting regions can be found in section 4.9.

4.4.3 *Review*

This section introduces a small language of sequences, variables, field access, and assignment. To type this language, we introduced the contexts Γ, which binds variables, and $\mathcal{H}$, which describes a reservation in terms of regions. We introduced the ability to explicitly track objects within regions in $\mathcal{H}$; this allows the manipulation of fields of these tracked objects with precise knowledge of the regions to which those fields pointed. Our efforts here were aided by an invariant: any field or object not explicitly tracked in $\mathcal{H}$ must be *simple*, meaning `isolated` fields refer to otherwise-inaccessible regions not currently tracked in $\mathcal{H}$. It is from this property that the `isolated` keyword gets its name.

Recognizing the desire to introduce and remove objects from precise tracking in $\mathcal{H}$, we introduced the virtual commands `focus`, `explore`, `retract`, and `unfocus`. These commands have no computational content; they manipulate the representation of $\mathcal{H}$, gaining or losing access to `isolated` fields and variables in the process. Recognizing that one may sometimes wish to connect non-`isolated` fields to objects in a different

region, we introduced the `attach` command to unify regions and make such assignments possible.

We proceed from here by expanding our language; first by adding functions, and then by adding the remaining familiar control structures of IMP.

4.5 ADDING FUNCTIONS

The previous section introduced a small language with objects and *virtual commands* capable of reflecting the structure of these objects into our typing environments $\mathcal{H}$ and Γ. It also included examples of non-*simple* structures that could be created within this language. At the time, we mentioned that there would be a cost to deviating from simplicity; that cost comes due with functions.

Programming with regions and tracked references requires reasoning about how the heap context $\mathcal{H}$ is changed during the execution of a function, and reasoning about the shape of regions provided to the function. This fine-grained reasoning conflicts with our goal of keeping programmer-facing complexity low; if we require programmers to annotate every function with the $\mathcal{H}$ required to call it and the $\mathcal{H}$ left after its execution, we will have overshot the desired complexity of our language.

We need to keep things simple. Simple regions have no interesting information tracked in $\mathcal{H}$; if Γ contains only simple variables, then the corresponding $\mathcal{H}$ is easy to generate. Even the names contained in $\mathcal{H}$ are arbitrary; they exist only to group objects by region. So by limiting function application to only simple objects, and ensuring that all results

returned from functions are themselves simple, we can avoid including any representation of $\mathcal{H}$ in the function signature at all.

For the remainder of this section we consider only single-argument functions with simple parameters. Focusing on single-argument functions allows us to grapple with the challenge of managing regions through abstraction without introducing peripheral complexity.

4.5.1 *New Keywords for Function Definition*

Section 4.2 introduced the example of a doubly-linked list each of whose `payload` nodes lived in a separate region. We motivated this choice by reasoning that one may wish to take an element of such a list and send it to another thread, thus removing it from the reservation. This scenario is illustrated in the following example:

Example 4.5.1. Sending away list contents.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle\ \rangle$
`focus x`
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle x[\ ]\rangle$
`explore x.payload`
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\ \rangle$
`send_to_thread(x.payload)`

????

What should the state of Γ and $\mathcal{H}$ be after the execution of `send_to_thread`? From its name we might guess that the `send_to_thread` function should consume its argument, removing its region from $\mathcal{H}$ and producing this output context:

$$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell']\rangle$$

And indeed, this is correct. But while we are free to make inferences from the name of our functions, the type system is not. The language needs new keywords that programmers can use to describe what effect function application has on the state of $\mathcal{H}$. Choosing to focus on functions which take and return only a single simple argument vastly simplifies[1] this task; the only possible function behavior is to remove its arguments' regions from $\mathcal{H}$, or to add its result's regions to $\mathcal{H}$. Effectively the only question is whether the function's result is in the same region as its argument, and whether the function's argument's region is lost during the execution of the function.

To accomplish this we introduce two new keywords: `consumes` and `preserves`. Both these keywords decorate arguments; the `consumes` keyword is applied to an argument whose region will not survive the function, while the `preserves` keyword is applied to an argument whose region *will* survive the function. Joining these keywords is a new use of the `isolated` keyword. When the `isolated` keyword decorates a function's return type, then that function's result lives in a separate region from its arguments. Conversely, when the keyword does not appear on the function return type, then the result of the function lives in the same region as its arguments.

Consider the following examples. First, consider the `send_to_thread` function called in example 4.5.1. Taking some liberties with the existence of a `void` type, the function signature of `send_to_thread` would look like this:

```
void send_to_thread(consumes T payload)
```

1 Pun intended.

Meanwhile, a standard print function—which should definitely not be damaging its arguments—would look like this:

```
void print(preserves T payload)
```

Branching out into functions which actually return something, the identity function's signature is:

```
T identity(preserves T t)
```

A clone function, which performs a deep copy of its argument, would both preserve its argument and place its result in a new region:

```
isolated T clone(preserves T t)
```

We believe that these annotations are both clear and easy to use; if a function does not intend to drop its argument from the current reservation, then it should be annotated `preserves`. If a function would like the liberty to drop its argument, then `consumes` is the correct annotation. If a function knows that its result will live in an otherwise-unreachable region, then the `isolated` keyword should be included; otherwise, it should be omitted.

4.5.2 *Adding Single-Argument Functions*

We now add functions with these annotations to our language. The syntax for functions and function application in this extension can be found in figure 4.12. Function names, represented by the metavariable *fname*, are drawn from a reserved namespace. Much like variable names, we've

$$
\begin{aligned}
p &::= q_r\ \tau\ \mathit{fname}(q\ \tau\ x)\{e\};\ p \mid e \\
e &::= \ldots \mid e(e) \mid \mathit{fname} \\
q &::= \texttt{consumes} \mid \texttt{preserves} \\
\tau &::= \ldots \mid (q\ \tau \to q_r\ \tau)
\end{aligned}
$$

Figure 4.12: Function definition and application syntax for ref-IMP. The *fname* is from a reserved set of identifiers which refer to functions.

chosen to make function names an expression of their own; this in turn means function application takes an expression in both the function and argument position. As discussed in subsection 4.5.1, function definitions include the new `isolated`, `consumes`, and `preserves` keywords; the `isolated` keyword is optional on the return result, while exactly one of the `consumes` or `preserves` keywords may appear on the function argument.

Semantically, we choose to represent our functions in a store-passing (rather than substitution) style. Function application behaves as assignment to the function parameter followed by a jump to its definition. Functions jump back to their point of application on return. As before, we make this choice to mirror the semantics of function application in Java, on which Gallifrey is based.

4.5.3 *Typing Rules for Functions*

The discussion so far has focused on what effect functions have on *regions*—whether they should consume or preserve their argument's region, and in what region their result should live. In our system, these are the interesting questions of function application; beyond these questions, our treatment of functions is a mild variation on the standard. We'll now focus on the

APPLY

$$\frac{\mathcal{H};\Gamma \vdash e_f : \ell_f\ (q\ \tau \to q_r\ \tau') \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma' \vdash e : \ell_e \tau \dashv \mathcal{H}'';\Gamma'' \qquad \Gamma'' \vdash (q\ \ell_e \to q_r\ \ell) : \mathcal{H}''' \Rightarrow \mathcal{H}_{out}}{\mathcal{H};\Gamma \vdash e_f(e) : \ell\tau \dashv \mathcal{H}_{out};\Gamma''}$$

LOOKUP

$$\frac{(\mathit{fname} : \tau) \in \mathcal{F} \qquad \ell\ \mathit{fresh}}{\mathcal{H};\Gamma \vdash \mathit{fname} : \ell\ \tau \dashv \Gamma;\mathcal{H}, \ell\langle\,\rangle}$$

DEFINE

$$\frac{\mathit{fname} : (q\ \tau \to q_r\ \tau') \in \mathcal{F} \qquad \ell\langle\,\rangle; x : \ell\ \tau \vdash e : \ell'\ \tau' \dashv \mathcal{H}_{f1}, \mathcal{H}_{f_2};\Gamma'_f \qquad \cdot \vdash (q\ \ell \to q_r\ \ell'') : \ell\langle\,\rangle \Rightarrow \mathcal{H}_{f1}}{\vdash q_r\ \tau'\ \mathit{fname}(q\ \tau\ x)\{e\}}$$

Figure 4.13: Function application and definition typing rules, written with reliance on the heap rules in figure 4.14.

typing components of the function rules, written with a reliance on the heap rules in figure 4.14 which is discussed later. The typing rules for functions can be found in 4.13.

We first discuss application. Recall that the form of application for this language has an expression in the function position; the application rule therefore first needs to check this expression and ensure it is a function. With the information from this function type, the rule ensures its argument's type matches the function's declared argument type, and the result of the application expression itself has the function's specified return type, as is standard. We have moved the interesting questions of function application—the effect on regions— to a heap rule, the third premise of APPLY.

Taking advantage of the simplicity of function arguments, the function definition rule first constructs a typing context $\Gamma = x : \ell\ \tau; \mathcal{H} = \ell\langle\,\rangle$ which contains a single variable x inside a simple region. As is standard, it then checks that the result type of the function body matches the declared

return type. Syntactically, functions are defined outside of an expression context, and have a name (*fname*) drawn from a fixed list of function name symbols. We also assume the existence of a global environment $\mathcal{F}$, which maps *fname*s to types. The function definition rule also ensures that functions have a name and type consistent with an entry in $\mathcal{F}$. It relies on heap rules to determine which region the function result should live in.

The LOOKUP rule handles the use of an *fname* as an expression. This rule consults $\mathcal{F}$ to determine the function type associated with *fname*, and creates a new simple region in which the function value will live. This creation of a new region may seem surprising; it semantically implies that the function object will be copied every time an *fname* expression appears. This is however a polite fiction; as function objects have no state, they are effectively never lost from the reservation and so this implied copy can be safely elided at runtime.

4.5.4 *Heap Rules for Functions*

Most of the interesting questions about functions—how they affect regions—have been deferred to the heap rules. The first interesting thing to note is that both the definition and application rule from figure 4.13 use the *same heap judgment* to determine the correct region for the function result. This judgment describes how the heap should change after applying a function, based largely on the keywords on the function's return and arguments. In its use in application, this rule describes the maximum assumptions that the application context can make about the output label and output heap context. In its use in definition, this rule describes the

$$cmd ::= \ldots \mid \texttt{consumes}\ \ell \mid \texttt{preserves}\ \ell$$

CONSUMES
$$\Gamma \vdash \texttt{consumes}\ \ell : \mathcal{H}, \ell\langle\rangle \Rightarrow \mathcal{H}$$

PRESERVES
$$\Gamma \vdash \texttt{preserves}\ \ell : \mathcal{H}, \ell\langle\rangle \Rightarrow \mathcal{H}, \ell\langle\rangle$$

PRESERVE-ISOLATED-FUNC
$$\frac{\Gamma \vdash q\ \ell : \mathcal{H} \Rightarrow \mathcal{H}' \qquad \ell' \notin \textit{region-names}(\Gamma, \mathcal{H}')}{\Gamma \vdash (q\ \ell \rightarrow \texttt{isolated}\ \ell') : \mathcal{H} \Rightarrow \mathcal{H}', \ell'\langle\,\rangle}$$

CONSUME-FUNC
$$\frac{\Gamma \vdash (\texttt{consumes}\ \ell \rightarrow \texttt{isolated}\ \ell') : \mathcal{H} \Rightarrow \mathcal{H}'}{\Gamma \vdash (\texttt{consumes}\ \ell \rightarrow \cdot\ \ell') : \mathcal{H} \Rightarrow \mathcal{H}'}$$

PRESERVE-SIMPLE-FUNC
$$\frac{\Gamma \vdash \texttt{preserves}\ \ell : \mathcal{H} \Rightarrow \mathcal{H}}{\Gamma \vdash (\texttt{preserves}\ \ell \rightarrow \cdot\ \ell) : \mathcal{H} \Rightarrow \mathcal{H}}$$

Figure 4.14: Function heap syntax and rules, with cases explicitly enumerated as separate rules. The *region-names* function produces a set of all region names mentioned by its arguments.

constraints to which the function definition must adhere. The rules for this judgment are found in figure 4.14.

Figure 4.14's five inference rules are an explicit casing of the keyword combinations that may appear on functions, and are best considered in two groups: the "argument" rules and the "execution" rules. The first two rules, CONSUMES and PRESERVES, describe how argument keywords affect the heap. Both rules require that the region in question is present and simple; the CONSUMES rule removes this region from the output heap context, while the PRESERVES rule leaves the output heap context unmodified.

The next three rules are the "execution" rules: these describe the result of function execution directly, using the argument rules in their premise. These rules are just an explicit casing of the possible keyword combinations in single-argument functions. Two rules concern functions which return an unannotated result, and one rule handles functions which produce

an `isolated` result. Interestingly the CONSUME-FUNC rule, which handles functions that `consume` their input region *without* producing an isolated result defers directly to the PRESERVE-ISOLATED-RESULT rule; in a single argument function, the output of a `consumes` function is *always* unrelated to it input as its input no longer exists by the function's end.

We now turn to the use of these heap rules in function application and definition. The strictest case is for functions which `preserve` their argument and produce an `isolated` result. At application time, the heap rules assume that the result lives in a new region, and so the definition must be checked under the same assumption. When defining this function, after the body executes there must exist two live regions: one in which the argument lives, and one in which the return result lives.

Next we turn to functions which preserve their argument and return a non-isolated result. Here the heap rule ensures that the argument's region is contained within the heap context, and that the result will live in the argument's region. This is conservative. As functions are opaque, the application rule must assume that the returned result is arbitrarily related to the argument. Function definition makes this same assumption; as all regions may be merged via `attach`, this choice does not result in a loss of expressivity.

We now discuss functions which `consume` their argument. Interestingly, one can *always* treat these functions as though they return an `isolated` result! This is because the function's signature guarantees that the region of its argument *will not survive* function application. On application, this means that any variables bound to the argument's region will become unusable, and that the function result will live in a new region. Function definition technically does not require anything in particular; the function

result needs to live in a region mentioned in neither an empty Γ nor an empty $\mathcal{H}$, and thus can be in any region.

4.5.5 *A Cavalcade of Examples*

To build intuition into how the function rules work in practice, we now present several examples of function definition and uses.

Example 4.5.2. Sending away list contents.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle\ \rangle$
`focus x`
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle x[\ ]\rangle$
`explore x.payload`
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell']\rangle, \ell'\langle\ \rangle$
`send_to_thread(x.payload)` *//send_to_thread : (consumes* $T \to$ *void)*
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle \texttt{x}[\texttt{payload} \mapsto \ell']\rangle$

To start, we revisit the example from the beginning of this section. Initially, a single `ListNode` x lives in a simple region. Focus and explore introduces a new region ℓ' in which `x.payload` lives. According to its type, `send_to_thread` consumes its argument and produces void. The application typing rule states that applying `send_to_thread` causes the argument's region—ℓ'—to be removed from $\mathcal{H}$. As a result of this, the list is no longer simple; we will be unable to `retract` away the tracked `payload` field, and will thus be unable to return the list from a function or pass it into some other function—at least until its simplicity is restored.

Example 4.5.3. Calling a clone.

$\Gamma = \texttt{x} : \ell\ \texttt{ListNode}, \texttt{y} : \ell\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle\ \rangle$
`y = clone(x)` *//clone : (preserves* $T \to$ *isolated* T*)*
$\Gamma = \texttt{x} : \ell\ \texttt{ListNode}, \texttt{y} : \ell'\ \texttt{ListNode} \quad \mathcal{H} = \ell\langle\ \rangle, \ell'\langle\ \rangle$

This is an example of a function which returns an `isolated` result. As a result of its execution, $\mathcal{H}$ gains a new region containing the result of `clone`. As `clone` preserves its arguments, `x`'s region also stays in $\mathcal{H}$.

Example 4.5.4. Defining identity.

```
T identity(preserves T o){
Γ = o : ℓ T   H = ℓ⟨ ⟩
 o
Γ = o : ℓ T   H = ℓ⟨ ⟩
}
```

This is a simple example of function definition: the identity function. The function definition rule states that functions begin with their parameter bound in a simple region; further, functions which `preserve` their region must ensure that it is simple at the end of function execution, as it is here. This function returns a non-isolated result; this result must therefore live in the same region as its argument (which it does).

Example 4.5.5. Cloning contents of a list.

```
isolated T get_head(preserves ListNode ln){
Γ = ln : ℓ ListNode   H = ℓ⟨ ⟩
 focus ln
Γ = ln : ℓ ListNode   H = ℓ⟨ln[ ]⟩
 explore ln.payload
Γ = ln : ℓ ListNode   H = ℓ⟨ln[payload ↦ ℓ']⟩,ℓ'⟨ ⟩
 ret = clone(ln.payload) //clone : (preserves T → isolated T)
Γ = ln : ℓ ListNode, ret : ℓ'' T   H = ℓ⟨ln[payload ↦ ℓ']⟩,ℓ'⟨ ⟩,ℓ''⟨ ⟩
 retract ln.payload
Γ = ln : ℓ ListNode, ret = ℓ''⟨ ⟩   H = ℓ⟨ln[ ]⟩,ℓ''⟨ ⟩
 unfocus ln
Γ = ln : ℓ ListNode, ret = ℓ''⟨ ⟩   H = ℓ⟨ ⟩,ℓ''⟨ ⟩
 ret OK: argument region and return region simple.
}
```

This is a more interesting function. It assumes the existence of variable definition, rules for which are introduced in the next section in figure 4.16.

As before, focus and explore gains access to `ln.payload` in region ℓ'. The type of clone is (`preserves` T $\to$ `isolated` T); according to the application rule, the argument's region persists after the clone and a new region is introduced to contain the `clone`'s result. The example cannot just stop at the point of the clone, however, because that would violate simplicity. This function `preserves` its input, promising that the input region will exist at the end of the function, *and* that it will be simple at that point. So the result of clone must be stored into a temporary variable, followed by a retract and unfocus to restore simplicity. In practice these virtual commands can be inferred.

Example 4.5.6. Using `get_head`.

$\Gamma =$ `x` $: \ell$ `ListNode`, `y` $: \ell$ `T` $\quad \mathcal{H} = \ell\langle\,\rangle$
`y = get_head(x)` *// get_head : (preserves ListNode* $\to$ *isolated T)*
$\Gamma =$ `x` $: \ell$ `ListNode`, `y` $: \ell'$ `T` $\quad \mathcal{H} = \ell\langle\,\rangle, \ell'\langle\,\rangle$

All the virtual commands required to clone the payload of the list element away are encapsulated within `get_head`; using it is now only a matter of ensuring the simplicity of its argument. What if one wanted to *remove* the payload from the list node instead?

Example 4.5.7. Negative example: `take_head`.

```
isolated T take_head(preserves ListNode ln){
Γ = ln : ℓ ListNode   H = ℓ⟨ ⟩
 focus ln
Γ = ln : ℓ ListNode   H = ℓ⟨ln[ ]⟩
 explore ln.payload
Γ = ln : ℓ ListNode   H = ℓ⟨ln[payload ↦ ℓ′]⟩, ℓ′⟨ ⟩
 ln.payload
Γ = ln : ℓ ListNode   H = ℓ⟨ln[payload ↦ ℓ′]⟩, ℓ′⟨ ⟩
ERROR: preserved region is non-simple
}
```

Example 4.5.7 attempts this, but it does not typecheck; at the end of this function body, its argument's region is not simple. Since example 4.5.7 `preserves` its argument's region, It must be simple when the function ends.

Example 4.5.8. Negative example: `take_head`, try 2.

```
isolated T take_head(preserves ListNode ln){
Γ = ln : ℓ ListNode   H = ℓ⟨ ⟩
 focus ln
Γ = ln : ℓ ListNode   H = ℓ⟨ln[ ]⟩
 explore ln.payload
Γ = ln : ℓ ListNode   H = ℓ⟨ln[payload ↦ ℓ′]⟩, ℓ′⟨ ⟩
 ret = ln.payload
Γ = ln : ℓ ListNode, ret = ℓ′⟨ ⟩   H = ℓ⟨ln[payload ↦ ℓ′]⟩, ℓ′⟨ ⟩
 retract ln.payload
Γ = ln : ℓ ListNode, ret = ℓ′⟨ ⟩   H = ℓ⟨ln[ ]⟩
 unfocus ln
Γ = ln : ℓ ListNode, ret = ℓ′⟨ ⟩   H = ℓ⟨ ⟩
 ret
ERROR: return expression region not in H
}
```

Unfortunately, naively following the same pattern as `get_head` will not work here; retracting `ln.payload` renders the `ret` variable inaccessible, preventing the function from returning it.

There is in fact no way to make this function work with a `preserves` argument and `isolated` result. A `preserves` $\rightarrow$ `isolated` function declares that its parameter and return result are disconnected; in the case of a list node and its `payload` field, they manifestly are connected. One could try removing the isolated result, declaring that the return result and argument live in the same region.

Example 4.5.9. Negative example: `take_head`, try 3.

```
isolated T take_head(preserves ListNode ln){
Γ = ln : ℓ ListNode   H = ℓ⟨ ⟩
 focus ln
Γ = ln : ℓ ListNode   H = ℓ⟨ln[ ]⟩
 explore ln.payload
Γ = ln : ℓ ListNode   H = ℓ⟨ln[payload ↦ ℓ']⟩, ℓ'⟨ ⟩
 ret = ln.payload
 attach ret to ln //necessary: ret and ln must be in the same region
Γ = ln : ℓ ListNode, ret = ℓ⟨ ⟩   H = ℓ⟨ln[payload ↦ ℓ]⟩
 retract ln.payload
Error: ℓ is not an empty region.
}
```

But this does not help. To write a function which preserves its input and returns a non-`isolated` result requires that the result of the function live in the same region as its argument. For this example, that means `ln` must live in the same region as `ln.payload`. This is a violation of simplicity; `ln.payload` is an `isolated` field, and `isolated` self-references are not simple. Delaying the point of `attach` later is impossible, because the moment the payload is retracted `ret` becomes inaccessible.

Example 4.5.10. Positive example: `take_head`, take 4.

```
isolated T take_head(consumes ListNode ln){
Γ = ln : ℓ ListNode   H = ℓ⟨ ⟩
 focus ln
Γ = ln : ℓ ListNode   H = ℓ⟨ln[ ]⟩
 explore ln.payload
Γ = ln : ℓ ListNode   H = ℓ⟨ln[payload ↦ ℓ']⟩, ℓ'⟨ ⟩
 ln.payload
Γ = ln : ℓ ListNode   H = ℓ⟨ln[payload ↦ ℓ']⟩, ℓ'⟨ ⟩
OK: argument region discarded, result region simple
}
```

The only way to make this example work is to have the function `consume` its argument. Now the function no longer needs to ensure that its argument's region is simple, though it still must ensure that the result's region is simple. This is a heavy price to pay; in returning the payload of the list node, we have lost access to the entire rest of the list.

4.6 FLESHING OUT THE LANGUAGE: ADDING IMP

With functions and structures in place, we have addressed the meat of our language. But to make it a realistic language we still have more to add; as a bare minimum, we will introduce the familiar constructs of IMP, gaining a way to declare variables, branch on conditionals, and loop.

4.6.1 *Syntax and Semantics of IMP*

The syntax of IMP, can be found in figure 4.15. Sticking with our Java-like target semantics, our IMP deviates from normal IMP slightly:

Figure 4.15: The syntax of IMP.

$$
\begin{aligned}
e &::= \ldots \mid \texttt{if}\ (e)\ \{e\}\ \texttt{else}\ \{e\} \mid \texttt{while}\ (e)\ \{e\} \mid e \oplus e \\
&\quad \mid \texttt{declare}\ x : \tau\ \texttt{in}\ \{e\} \mid n \mid \texttt{true} \mid \texttt{false} \\
\oplus &::= \ ; \mid + \mid * \mid - \mid \texttt{\&\&} \mid \texttt{||} \\
\tau &::= \ldots \mid \texttt{int} \mid \texttt{bool} \mid \texttt{void}
\end{aligned}
$$

- We have chosen not to separate statements and expressions. As commonly accompanies this choice, we'll have conditionals and sequences return the result of their last-executed subexpression, and we'll have while-loops explicitly return void.
- We have chosen to have explicit variable declarations with lexical scope.
- Our variable declarations are not initialized, and must be assigned to before use.

The features we include from IMP are otherwise straightforward.

4.6.2 *Typing Rules for IMP*

The additional typing rules we require for IMP can be found in figure 4.16. The CONDITIONAL and WHILE rules are, as always, the most interesting rules here. A conditional branch or loop body may modify its heap context (for example by assignment to an `isolated` field) or modify its typing context (for example by assigning a variable to an object from a different region). In the case of the conditional, such modifications to our environment can only be kept if an equivalent modification occurred in the other branch. For the

CONSTANT

$$\frac{\ell \textit{ fresh}}{\mathcal{H};\Gamma \vdash n : \ell\ \texttt{int} \dashv \mathcal{H}, \ell\langle\,\rangle;\Gamma}$$

BOOL-CONSTANT

$$\frac{b \in \{\texttt{true}, \texttt{false}\} \qquad \ell \textit{ fresh}}{\mathcal{H};\Gamma \vdash b : \ell\ \texttt{bool} \dashv \mathcal{H}, \ell\langle\,\rangle;\Gamma}$$

DECLARE

$$\frac{x \notin \Gamma \qquad x \notin \mathcal{F} \qquad \mathcal{H};\Gamma, x : \bot\ \tau \vdash e : \ell\ \tau' \dashv \mathcal{H}';x : \ell'\tau,\Gamma'}{\mathcal{H};\Gamma \vdash \texttt{declare}\ x : \tau\ \texttt{in}\{e\} : \ell\ \tau' \dashv \mathcal{H}';\Gamma'}$$

INFIX

$$\frac{\mathcal{H};\Gamma \vdash e_1 : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma' \vdash e_2 : \ell\ \tau \dashv \mathcal{H}'';\Gamma'' \qquad \vdash \tau \oplus \tau}{\mathcal{H};\Gamma \vdash e_1 \oplus e_2 : \ell\ \tau \dashv \mathcal{H}'';\Gamma''}$$

CONDITIONAL

$$\frac{\begin{array}{c}\mathcal{H};\Gamma \vdash e_1 : \ell_b\ \texttt{bool} \dashv \mathcal{H}';\Gamma' \\ \mathcal{H}';\Gamma' \vdash e_2 : \ell_l\tau \dashv \mathcal{H}_l, \mathcal{H}_2;\Gamma_l,\Gamma_2 \qquad \mathcal{H}';\Gamma' \vdash e_3 : \ell_r\tau \dashv \mathcal{H}_r, \mathcal{H}_3;\Gamma_r,\Gamma_3 \\ \mathcal{H}_l;\Gamma_l \equiv_m \mathcal{H}_r;\Gamma_r \qquad m(\ell_l) = \ell_r \qquad \mathit{vars}(\Gamma) = \mathit{vars}(\Gamma_l)\end{array}}{\mathcal{H};\Gamma \vdash \texttt{if}(e_1)\ \{e_2\}\ \texttt{else}\ \{e_3\} : \ell_l\ \tau \dashv \mathcal{H}_l;\Gamma_l}$$

WHILE

$$\frac{\begin{array}{c}\mathcal{H};\Gamma \vdash e_1 : \ell_b\ \texttt{bool} \dashv \mathcal{H}';\Gamma' \\ \mathcal{H}';\Gamma' \vdash e_2 : \ell\ \tau \dashv \mathcal{H}_l, \mathcal{H}_2;\Gamma_l,\Gamma_2 \qquad \mathcal{H}';\Gamma' \equiv_m \mathcal{H}_l;\Gamma_l\end{array}}{\mathcal{H};\Gamma \vdash \texttt{while}(e_1)\ \{e_2\} : \bot\ \texttt{void} \dashv \mathcal{H}';\Gamma'}$$

Figure 4.16: Typing rules for ref-IMP.

loop, such modifications can only be kept if the resulting environments are equivalent to the loop's initial environments. This equivalence is decided by the $\equiv_m$ equivalence relation, defined in figure 4.17. This equivalence relation relates two environment pairs which differ only in the precise names chosen for their regions. This captures the fact that our region labels are just arbitrary symbols; the names were initially chosen arbitrarily, and so renaming regions is harmless so long as the new labeling does not change the structure of $\mathcal{H}$ or the validity of variables in Γ. It's important to note here that the m subscript is not a marker; it's the *actual function* used to rename the left-hand side into the right-hand side. The CONDITIONAL rule takes advantage of this, ensuring that the labels on the result of both branches are equivalent under the same m used to determine the branches' environment equivalence. Of final note, both the WHILE and CONDITIONAL rules discard any parts of the environment that are not equivalent, partitioning them into environments which do not propagate to the rule's output, so long as all variable bindings from the input Γ are preserved.

The remaining rules are mostly straightforward; of note is that constants get a new region at each use, and the behavior of the DECLARE rule. The DECLARE rule does not initialize the declared variable to a particular expression; we capture this lack of initialization by setting its region label to $\perp$, the inaccessible region. Assignment overwrites this with a real region, following the assignment rule introduced in section 4.3.

REGION-α-EQUIVALENCE

$$\frac{m = (\ell_1 \mapsto \ell'_1, \ldots, \ell_n \mapsto \ell'_n) \text{ is a bijection from } \mathit{regions}(\mathcal{H}) \text{ to } \mathit{regions}(\mathcal{H}') \qquad \mathcal{H} = \ell\langle x[f \mapsto \ell'', \ldots], \ldots\rangle, \ldots}{\mathcal{H}, \Gamma \equiv_m m(\ell)\langle x[f \mapsto m(\ell''), \ldots], \ldots\rangle, \ldots; \{x : m(\ell)\tau \mid x : \ell\tau \in \Gamma\}}$$

Figure 4.17: Environment equivalence for ref-IMP.

$$\begin{aligned} v &::= l \mid \mathit{constant} \\ e &::= \ldots \mid v \end{aligned}$$

Figure 4.18: New syntax for values and locations.

4.7 A DYNAMIC SEMANTICS

We now introduce a formal dynamic semantics of the language.

4.7.1 *Adding Locations*

The language requires a few additions to support a small-step semantics and the accompanying progress and preservation proofs. First, we explicitly identify a set of irreducible values v, including a new *location* expression l (figure 4.18). We need to introduce locations because we're

$$(l : \ell\tau) \in \mathrm{P}$$

LOCATION-REFERENCE

$$\mathrm{P}, l : \ell\ \tau \vdash l : \ell\ \tau$$

LOCATIONS

$$\frac{\mathrm{P} \vdash l : \ell\ \tau \qquad \mathcal{H} \vdash \ell}{\mathcal{H}; \Gamma; \mathrm{P} \vdash l : \ell\ \tau \dashv \mathcal{H}; \Gamma}$$

Figure 4.19: A typing rule for locations.

modeling a Java-like semantics where aliasing is important; in our language, all variables contain *references* and the expression $x = y$ means "set x to the same reference as y". The small-step semantics steps the expression $x = y$ to some expression $x = l$; so we'll need an expression form for these l, as well as a typing rule for them.

Locations can never appear in the surface syntax of a program, and will only arise as the result of some step in a small-step evaluation. This in turn means that the existing typing contexts have no way to type locations; introducing a typing rule for locations requires also introducing a new *store typing* environment holding locations' types. This environment is P; its form and new typing rules using it can be found in figure 4.19.

We'll also need to thread P through the rest of the typing rules, passing it recursively to subexpressions as they are visited in each rule's premises. This modification is reflected in the full system found in the appendix; we choose not to extract it and highlight it here. One aspect of P that is perhaps odd at first glance is that it, unlike Γ and $\mathcal{H}$, never appears on the output of a typing rule and is never modified by any typing rule. This is an artifact of how we have phrased progress and preservation; we will not need to adjust the mapping from locations to regions or types.

4.7.2 *Communication*

To model gaining and losing objects from the reservations, we will introduce abstract communication primitives `send` and `receive`. The extended syntax and typing rules for them can be found in figure 4.20. We leave abstract the destination to which `send` sends and from which `receive`

$$e ::= \ldots \mid \texttt{send-}\tau \mid \texttt{receive-}\tau$$

$$\mathcal{H}; \Gamma; \mathrm{P} \vdash \texttt{send-}\tau : (\texttt{consumes } \tau \to \texttt{void}) \dashv \Gamma; \mathcal{H}$$

$$\mathcal{H}; \Gamma; \mathrm{P} \vdash \texttt{receive-}\tau : (\texttt{void} \to \texttt{isolated } \tau) \dashv \Gamma; \mathcal{H}$$

Figure 4.20: Send and receive communication primitives.

$$\begin{aligned} R_d &: 2^l \\ \pi &: l \rightharpoonup o \\ o &: (\tau, v) \\ \sigma &: x \rightharpoonup l \end{aligned}$$

Figure 4.21: Definitions for our smallstep configuration elements.

receives; we mean only for these to stand in for arbitrary communication, and to model their effect on the reservation. We assume there is a pair of `send` and `receive` functions for each type.

4.7.3 *Small Steps*

Recall from section 4.2 that our purpose is to prove reservation safety: that, given a set of objects which a context may access, evaluation only ever accesses objects in that set. We choose to present a single-threaded dynamic semantics in which reservations are explicitly tracked, and prove that no thread accesses memory outside their reservation. This core result can then be used to prove memory safety (where reservations represent allocated memory), thread safety (where each thread has a reservation and no reservation may overlap), or plain isolation as needed.

$$
\begin{aligned}
E ::= [\cdot] \;&|\; E \oplus e \mid l \oplus E \mid E; e \mid l; E \mid \texttt{if}\ (E)\ \{e\}\ \texttt{else}\ \{e\} \\
&|\; \texttt{while}\ (E)\ \{e\} \mid x = E \mid E(e) \mid l(E) \mid E.f \\
&|\; \texttt{send}\ E \mid \texttt{receive}\ E \mid \texttt{attach}\ E\ e \mid \texttt{attach}\ l\ E \\
&|\; \texttt{focus}\ E \mid \texttt{unfocus}\ E \mid \texttt{explore}\ E \mid \texttt{retract}\ E
\end{aligned}
$$

$$
\frac{(R_d, \pi, \sigma, e) \xrightarrow{eval} (R_d', \pi', \sigma', e')}{(R_d, \pi, \sigma, E[e]) \xrightarrow{eval} (R_d', \pi', \sigma', E[e'])}
$$

Figure 4.22: evaluation contexts for our language.

The small-step semantics for our language is in figure 4.23, with configurations defined in figure 4.21 and evaluation contexts in 4.22. Our semantics is presented as a relation $\xrightarrow{eval}$ between two configurations. Configurations are a 4-tuple of the form (R_d, π, σ, e), where R_d, a set of locations, represents the *reservation*, π, a partial function from locations to objects, represents the *heap*, σ, a partial function from variables to locations, represents the stack, and e remains an expression. One immediate thing to note here is that π represents the *entire* heap, not just the portion contained within the reservation; π restricted to keys in R_d is the set of available objects.

The idea behind this small-step semantics is that whenever stepping e would require looking at a specific location, we first check R_d to make sure that location is in the dynamic reservation. If it is, evaluation can step. If not, then evaluation gets stuck. Using this, we can reduce the proof of reservation safety to a proof of progress and preservation.

There are a few elements of this small-step semantics (beyond the presence of the reservation R_d) that are not quite standard. These are done

$$\pi \vdash l \hookrightarrow l \qquad \frac{\pi(l) = (\tau, v) \qquad v[f] = l'}{\pi \vdash l \hookrightarrow l'} \qquad \frac{\pi \vdash l \hookrightarrow l' \qquad \pi \vdash l' \hookrightarrow l''}{\pi \vdash l \hookrightarrow l''}$$

$$\frac{\sigma(x) \in R_d}{(R_d, \pi, \sigma, x) \xrightarrow{eval} (R_d, \pi, \sigma, \sigma(x))}$$

$$\frac{l \; fresh}{(R_d, \pi, \sigma, constant) \xrightarrow{eval} (R_d \cup \{l\}, \pi[l \mapsto (typeof(constant), constant)], \sigma, l)}$$

$$\frac{l_l, l_r \in R_d \qquad \pi(l_l) = o_l}{(R_d, \pi, \sigma, l_l; l_r) \xrightarrow{eval} (R_d, \pi, \sigma, l_r)} \qquad \frac{l_l, l_r \in R_d \qquad \pi(l_l) = o_l \qquad \pi(l_r) = o_r}{(R_d, \pi, \sigma, l_l \oplus l_r) \xrightarrow{eval} (R_d, \pi, \sigma, [\![\oplus]\!](o_l, o_r))}$$

$$\frac{l \in R_d \qquad \pi(l) = \texttt{true}}{(R_d, \pi, \sigma, \texttt{if}(l)\{e\}\ \texttt{else}\ \{e_i\}) \xrightarrow{eval} (R_d, \pi, \sigma, e)}$$

$$\frac{l \in R_d \qquad \pi(l) = \texttt{false}}{(R_d, \pi, \sigma, \texttt{if}(l)\{e_i\}\ \texttt{else}\ \{e\}) \xrightarrow{eval} (R_d, \pi, \sigma, e)}$$

$$(R_d, \pi, \sigma, \texttt{while}\ (e_1)\ \{e_2\}) \xrightarrow{eval} (R_d, \pi, \sigma, \texttt{if}(e_1)\ \{e_2; \texttt{while}\ (e_1)\ \{e_2\}\})$$

$$(R_d, \pi, \sigma, \texttt{declare}\ x : \tau\ \texttt{in}\ e) \xrightarrow{eval} (R_d, \pi, \sigma, e) \qquad (R_d, \pi, \sigma, x = l) \xrightarrow{eval} (R_d, \pi, \sigma[x \mapsto l], l)$$

$$\frac{l_f \in R_d \qquad \pi(l_f) = (\tau_f, v_f) \qquad F_d(v_f) = \lambda x.e \qquad e \equiv_\alpha e' \qquad FV(e') = \{x'\} \qquad x' \; fresh}{(R_d, \pi, \sigma, l_f(l)) \xrightarrow{eval} (R_d, \pi, \sigma, \{x' = l; e'\})}$$

$$\frac{dom(\pi') = R'_d \qquad \vdash \pi' \qquad \pi'(l) = (\tau, v) \qquad \forall l' \in dom(\pi').\ \ \pi' \vdash l \hookrightarrow l'}{(R_d \uplus R'_d, \pi \uplus \pi', \sigma, \texttt{send}\ \tau(l)) \xrightarrow{eval} (R_d, \pi \uplus \pi', \sigma, l)}$$

$$\frac{dom(\pi') \cap R_d = \emptyset \qquad \vdash \pi' \qquad \pi'(l) = (\tau, v) \qquad \forall l' \in dom(\pi').\ \ \pi' \vdash l \hookrightarrow l'}{(R_d, \pi \uplus \pi', \sigma, \texttt{receive}\ \tau()) \xrightarrow{eval} (R_d \cup dom(\pi'), \pi \uplus \pi', \sigma, l)}$$

$$\frac{l \in R_d \qquad \pi(l) = (\tau, v) \qquad v[f] = l_2}{(R_d, \pi, \sigma, l.f) \xrightarrow{eval} (R_d, \pi, \sigma, l_2)}$$

$$\frac{l \in R_d \qquad \pi(l) = (\tau, v) \qquad q_r\ f \in fields(\tau)}{(R_d, \pi, \sigma, l.f = l_2) \xrightarrow{eval} (R_d, \pi[l \mapsto (\tau, v[f \mapsto l_2])], \sigma, l_2)}$$

$$\frac{cmd \in \{\texttt{attach}\ l\ l_i, \texttt{focus}\ l, \texttt{unfocus}\ l, \texttt{explore}\ l, \texttt{retract}\ l\}}{(R_d, \pi, \sigma, cmd) \xrightarrow{eval} (R_d, \pi, \sigma, l)}$$

$$\begin{aligned}
[\![+]\!]((\texttt{int}, n_1), (\texttt{int}, n_2)) &\triangleq n_1 + n_2 \\
[\![*]\!]((\texttt{int}, n_1), (\texttt{int}, n_2)) &\triangleq n_1 * n_2 \\
[\![-]\!]((\texttt{int}, n_1), (\texttt{int}, n_2)) &\triangleq n_1 - n_2
\end{aligned}$$

Figure 4.23: A small-step semantics for ref-IMP + functions + structures.

to streamline the presentation. The first is that all expressions evaluate to a location; even constants, which are placed in the heap at a freshly-generated location. Because of this choice, evaluation contexts (figure 4.22) also step expressions until they reach locations, and conditional and looping constructs take locations as their condition, explicitly looking up the value at that location as part of stepping into the condition or loop's body. Declarations are effectively ignored; as we've made it a static requirement to avoid shadowing, have no closures, and require an assignment before reading a variable, we can bind variables to locations on assignment and directly read them on access without reasoning about scope. Correspondingly, we've chosen to implement function application as an α-rename of the function's free variable followed by an assignment to that variable and the execution of the function body.

Finally our `send` and `receive` execution here is of note. The `send` rule says that sending a location causes all objects reachable from that location to be removed from the reservation. The premise $\vdash \pi$, defined in figure 4.24, says that π is sane—all fields point to objects of the fields' type, and there are no dangling references. The premise $\pi \vdash l \hookrightarrow l'$ says that l' is reachable from l in π. Taken together, they mean that π is the complete object graph for some l, and contains no extraneous elements. The `receive` rule functions similarly, except here the object graph to receive is not bound by the syntax, but rather can be arbitrary.

4.8 CORRECTNESS

With the introduction of our dynamic semantics, we'll be able to prove *reservation safety*—the property that a step of execution only accesses objects within its reservation—via a standard progress and preservation proof. This is because our dynamic semantics "gets stuck" when it would need to access memory outside of its reservation; therefore if the semantics never "gets stuck", it never attempts such an access. But before we can state progress and preservation, we will need a good number of auxiliary definitions and to formalize several "intuitive" properties of our typing contexts.

4.8.1 *Store Typing*

Proving progress and preservation requires a typing on configurations: we need some way to say $\vdash (R_d, \pi, \sigma, e)$. This in turn requires a typing on stores—some way of saying that the static $\mathcal{H}$;Γ;P corresponds to the dynamic π, σ. This gets a bit complicated because the division of roles among π and σ is not directly mirrored by $\mathcal{H}$, Γ, and P. To begin with, regions actually aren't directly represented in π and σ at all; regions are a purely static abstraction, used to group objects whose non-`isolated` fields refer to each other. Static contexts, however, link to each other using regions; for example, Γ maps variables (tracked in σ) to types (tracked in π), and regions (tracked in P). In this subsection, we introduce definitions describing what it means for each static environment to correctly describe the heap and stack.

$$\frac{\begin{array}{c}\forall l \in dom(\pi), \tau, v \text{ s.t. } l = (\tau, v) \text{: } \forall f, l' \text{ s.t. } v[f] = l' \text{:} \\ \exists q_r, \tau_f, v_f \text{ s.t. } q_r\ f\ \tau_f \in fields(\tau) \wedge \pi(l') = (\tau_f, v_f)\end{array}}{\vdash \pi}$$

Figure 4.24: A well-formedness condition on π: objects' fields refer to other extant objects of the correct type.

$$\frac{\pi(l) = (\tau, v) \qquad q_r\ f \in fields(\tau) \qquad v[f] = l'}{\pi \vdash l[f] = l'}$$

$$\frac{\begin{array}{c}\forall l : (l \in dom(\pi)) \Rightarrow \exists \ell, \tau, v\text{: } (\mathrm{P} \vdash l : \ell\ \tau) \wedge (\pi(l) = (\tau, v)) \\ \forall l, l', f\text{: } (\pi \vdash l[f] = l') \Rightarrow \exists \ell, \tau, \tau'\text{: } \mathrm{P} \vdash l : \ell\ \tau \wedge ((\mathrm{P} \vdash l' : \ell\ \tau') \vee (\texttt{isolated}\ f : \tau' \in fields(\tau)))\end{array}}{\vdash \pi : \mathrm{P}}$$

Figure 4.25: P corresponds to π when all objects in π are mapped in P, and where only `isolated` references connect objects in distinct regions.

4.8.1.1 *A sane heap*

The first correspondences we will consider relate to the accurate mapping of objects in π. To begin, we need a sanity condition on π itself; we only want to consider πs where all fields map to objects of the appropriate type, and where no fields contain dangling pointers. This is expressed in 4.24.

The first environment typing will govern the correspondence between π and P, found in figure 4.25. Because P will only ever be generated directly from π in the process of proving progress and preservation, we can afford to make its correspondence with π very tight; in fact we require that all keys in π are given a label and type in P, and that this type matches the type found in π. This is also where one of the key *region invariants* is expressed: Only *isolated* references may connect objects in distinct regions.

$$\frac{\forall l, \tau \; s.t. \; \mathrm{P} \vdash l : \ell \; \tau\text{:} \;\; \pi; \mathrm{P} \vdash l \; \texttt{simple} \qquad \vdash \pi : \mathrm{P}}{\pi; \mathrm{P} \vdash \ell \; \texttt{simple}}$$

$$\frac{\begin{array}{c} \pi(l) = (\tau, v) \qquad \texttt{isolated}\; f \in \tau \\ \mathrm{P} \vdash l : \ell \; \tau \qquad \mathrm{P} \vdash v[f] : \ell' \; \tau' \qquad \ell \neq \ell' \qquad \pi; \mathrm{P} \vdash \ell' \; \texttt{dominated} \\ \pi; \mathrm{P} \vdash \ell' \; \texttt{simple} \qquad \pi; \mathrm{P} \vdash \ell' \; \texttt{no-cycles} \qquad \vdash \pi : \mathrm{P} \end{array}}{\pi; \mathrm{P} \vdash l.f \; \texttt{simple}}$$

$$\frac{\pi(l) = (\tau, v) \qquad \cdot f \in \tau}{\pi; \mathrm{P} \vdash l.f \; \texttt{simple}}$$

$$\frac{\pi(f) = (\tau, v) \qquad \forall q_r \; f \in \mathit{fields}(\tau)\text{:} \;\; \pi; \mathrm{P} \vdash l.f \; \texttt{simple}}{\pi; \mathrm{P} \vdash l \; \texttt{simple}}$$

Figure 4.26: Simplicity captures the idea of a forest; all isolated references from a simple object dominate their region, and the regions reachable from simple objects' `isolated` references must themselves be simple.

We say that any label-typing which adheres to these properties correctly types π.

4.8.1.2 *Simple regions, objects, and fields*

The section on objects relies heavily on the intuition that a *simple* region is one whose region graph forms a forest. This idea is captured in figure 4.26. We'll rely on this when formalizing the simplicity invariant on $\mathcal{H}$: that all objects, regions, and fields which *aren't* explicitly tracked in $\mathcal{H}$ must be simple.

Included in the notion of simplicity is the idea that a simple `isolated` reference should *dominate* its reachable object graph.

Definition 4.8.1 (Domination)**.** π;$\mathrm{P} \vdash \ell$ `dominated` if:

$\forall l_1, l_2, l_3, l_4$ where :

$$\begin{aligned}
&\mathrm{P} \vdash l_1 : \ell_1 \tau_1 && \text{for some } \tau_1 \text{ and } \ell_1 \neq \ell \\
\wedge &\mathrm{P} \vdash l_2 : \ell \tau_2 && \text{for some } \tau_2 \\
\wedge &\mathrm{P} \vdash l_3 : \ell_3 \tau_3 && \text{for some } \tau_3 \text{ and } \ell_3 \neq \ell \\
\wedge &\mathrm{P} \vdash l_4 : \ell \tau_4 && \text{for some } \tau_4 \\
\wedge &\pi \vdash l_1[f_1] = l_2 && \text{for some } f_1 \\
\wedge &\pi \vdash l_3[f_3] = l_4 && \text{for some } f_3
\end{aligned}$$

Then $l_1 = l_3 \wedge f_1 = f_3$.

This definition states that if there are any two external objects which refer to objects in some region, then those two external objects must be the same, and the field by which they refer to their target must also be the same. This expresses the property that there is at most one reference from outside the region into it.

Simplicity also requires that we reason about the absence of cycles in the region graph; this is captured by the following definition (which relies on on a definition of "region reachability" in figure 4.27):

Definition 4.8.2 (Cycle-free)**.**

$\pi; \mathrm{P} \vdash \ell$ `no-cycles` if $\forall l, l', \ell, \ell', \tau, \tau'$ where

$$\mathrm{P} \vdash l : \ell\ \tau \wedge \mathrm{P} \vdash l' : \ell'\ \tau' \wedge \mathrm{P}; \pi \vdash l \text{ -> } l' :$$

$$(\mathrm{P}; \pi \vdash l' \text{ -> } l) \Rightarrow (\ell = \ell')$$

$$\frac{\pi(l) = (\tau, v) \qquad \mathrm{P} \vdash l : \ell\ \tau}{\mathrm{P}; \pi \vdash l : \ell\ \tau} \qquad \frac{\mathrm{P}; \pi \vdash l : \ell\ \tau \qquad \mathrm{P}; \pi \vdash l' : \ell\ \tau'}{\mathrm{P}; \pi \vdash l\ \texttt{->}l'}$$

$$\frac{\begin{array}{c}\mathrm{P}; \pi \vdash l : \ell\ \tau \qquad \mathrm{P}; \pi \vdash l' : \ell'\ \tau' \\ \pi(l) = (\tau, v) \qquad v[f] = l' \qquad \texttt{isolated}\ f : \tau' \in \mathit{fields}(\tau)\end{array}}{\mathrm{P}; \pi \vdash l\ \texttt{->}l'}$$

$$\frac{\mathrm{P}; \pi \vdash l\texttt{->}l' \qquad \mathrm{P}; \pi \vdash l'\texttt{->}l''}{\mathrm{P}; \pi \vdash l\texttt{->}l''}$$

Figure 4.27: Definition of region reachability, which is reachability where all objects are reachable from all other objects in the same region.

$$\frac{\begin{array}{c}\sigma(x) = l \qquad \mathrm{P} \vdash l : \ell\ \tau \\ \pi(l) = (\tau, v) \qquad \forall f \mapsto \ell' \in F\colon (f \in \mathit{fields}(\tau)) \wedge \mathrm{P} \vdash v[f] : \ell' \\ \forall f \in (\mathit{fields}(\tau) - \mathit{fields}(F))\colon \pi; \mathrm{P} \vdash v[f]\ \texttt{simple} \\ \forall l' \neq l\colon (\mathrm{P} \vdash l' : \ell\ \tau') \Rightarrow (\pi; \mathrm{P} \vdash l'\ \texttt{simple}) \qquad \sigma, \mathrm{P} \vdash \pi : \mathcal{H} \qquad \vdash \pi : \mathrm{P}\end{array}}{\sigma, \mathrm{P} \vdash \pi : \ell\langle x[F]\rangle, \mathcal{H}}$$

$$\frac{\forall l\colon (\mathrm{P} \vdash l : \ell\ \tau) \Rightarrow (\pi; \mathrm{P} \vdash l\ \texttt{simple}) \qquad \sigma, \mathrm{P} \vdash \pi : \mathcal{H} \qquad \vdash \pi : \mathrm{P}}{\sigma, \mathrm{P} \vdash \pi : \ell\langle\rangle, \mathcal{H}}$$

Figure 4.28: Correspondence between π, P and $\mathcal{H}$, showing that all objects, regions, and fields not explicitly tracked in $\mathcal{H}$ must be simple.

$$\frac{\vdash \pi : \mathrm{P}}{\mathrm{P} \vdash (\pi, \sigma) : \cdot} \qquad \frac{\vdash \pi : \mathrm{P} \qquad \vdash (\pi, \sigma) : \Gamma \qquad \mathrm{P} \vdash \sigma(x) : \ell\ \tau}{\mathrm{P} \vdash (\pi, \sigma) : (x : \ell\ \tau, \Gamma)}$$

Figure 4.29: Demonstrating that Γ accurately represents σ and π.

4.8.1.3 *Simplicity in $\mathcal{H}$*

In figure 4.28 we see the payoff of our work defining simplicity: the simplicity invariant stated formally. Here we rely on the exact correspondence between π and P quite heavily; while we phrase this property as $\mathcal{H}$ corresponding to π under σ and P, we in practice rely more on P than on π. This recursive definition ensures three things. First, all objects in a region which *aren't* tracked are simple. Second, all untracked fields of tracked objects are simple. Finally, all tracked fields of tracked objects refer to the same region in P as they do in $\mathcal{H}$.

4.8.1.4 *Γ corresponds to σ and π*

Our last separate judgment form, found in figure 4.29, demonstrates that Γ accurately reflects the dynamic store: that it maps objects to types and regions consistent with σ and π. In this rule we once again rely on the direct correspondence between π and P, using the $\mathrm{P} \vdash l$ judgment exclusively to reason about the type of a location.

4.8.1.5 *Putting it all together*

With the independent definitions of correspondence, we can now finally define what it means for a dynamic heap and stack to be represented by Γ, $\mathcal{H}$, and P. This is found in figure 4.30.

$$\frac{\vdash \pi \qquad \vdash \mathcal{H};\Gamma \qquad \vdash \pi : \mathrm{P} \qquad \mathrm{P} \vdash (\pi, \sigma) : \Gamma \qquad \sigma, \mathrm{P} \vdash \pi : \mathcal{H}}{\vdash \pi, \sigma : \mathcal{H};\Gamma;\mathrm{P}}$$

Figure 4.30: Our store typing.

$$\frac{\forall \ell \text{ s.t. } \mathcal{H} \vdash \ell \,.\, \forall l, \tau \text{ s.t. } \mathrm{P} \vdash l : \ell\ \tau \,.\, l \in R_d \qquad \vdash \pi : \mathrm{P} \qquad R_d \subseteq dom(\pi)}{\pi; \mathrm{P} \vdash R_d : \mathcal{H}}$$

Figure 4.31: Our static reservation models our dynamic reservation.

4.8.2 *Approximating Dynamic Reservations*

We next ensure that the dynamic reservation R_d is correctly modeled by $\mathcal{H}$. This condition, found in figure 4.31, effectively states that all objects in a region permitted by $\mathcal{H}$ must be available in the reservation R_d. Note that $\mathcal{H}$ need *not* precisely correspond with R_d; it is acceptable to have extra statically inaccessible locations. In this definition we again rely on the precision of P; here, it is essential that all locations are mapped in P lest we miss some inaccessible members of an otherwise-accessible region.

4.8.3 *Putting it all Together: Configuration Typing*

With these components we can finally state what it means for a configuration to be well-typed, found in figure 4.32. As is standard for a configuration-based small-step semantics, we say that the configuration is well-typed if its expression is well-typed under a static typing environment corresponding to its dynamic store. This is in turn well-typed if the heap and stack are typed at $\mathcal{H}$;Γ;P, and if the static reservation in $\mathcal{H}$

$$\frac{\vdash \pi, \sigma : \mathcal{H}; \Gamma; \mathrm{P} \qquad \pi; \mathrm{P} \vdash R_d : \mathcal{H}}{\vdash (R_d, \pi, \sigma) : (\mathcal{H}; \Gamma; \mathrm{P})}$$

$$\frac{\vdash (R_d, \pi, \sigma) : (\mathcal{H}; \Gamma; \mathrm{P}) \qquad \mathcal{H}; \Gamma; \mathrm{P} \vdash e \dashv \Gamma'; \mathcal{H}'}{\vdash (R_d, \pi, \sigma, e)}$$

Figure 4.32: Definition of a well-typed configuration.

accurately models the dynamic reservation R_d for this particular heap and heap typing.

4.8.4 *Progress and Preservation*

We are now, finally, able to state progress and preservation. Preservation is where the invariants are hiding; in order to ensure an expression remains typeable after taking a step, we must ensure that this step preserves the invariants required by our static environments. Progress is where reservation safety is hiding; as we have written a dynamic semantics which cannot take a step that would violate its reservation, proving that our type system guarantees progress in turn ensures that the typed program cannot race.

Theorem 4.1 (Progress)**.** *For any* (R_d, π, σ, e) *where* e *is not a value, if* $\vdash (R_d, \pi, \sigma, e)$ *then there exists some* $(R'_d, \pi', \sigma', e')$ *such that* $(R_d, \pi, \sigma, e) \xrightarrow{eval} (R'_d, \pi', \sigma', e')$

Theorem 4.2 (Preservation)**.** *For any* (R_d, π, σ, e)*, if* $\vdash (R_d, \pi, \sigma, e)$ *and there exists some* $(R'_d, \pi', \sigma', e')$ *where* $(R_d, \pi, \sigma, e) \xrightarrow{eval} (R'_d, \pi', \sigma', e')$ *then* $\vdash (R'_d, \pi', \sigma', e')$

$$\textsc{detach}\quad \frac{\mathcal{H};\Gamma;\mathrm{P} \vdash e : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{retract}\ \ell : \mathcal{H}' \Rightarrow \mathcal{H}'' \qquad \ell'\ \textit{fresh}}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{detach}\ e : \ell'\ \tau \dashv \mathcal{H}'', \ell'\langle\,\rangle;\Gamma'}$$

Figure 4.33: Typing rules for `detach`.

Note that these theorems have not been through a rigorous syntactic proof.

4.9 EXTENSION: DETACH

Among the virtual commands, the `attach` command stands out: it has no natural inverse. This is because `attach` "forgets" that the objects previously contained in the attaching region are disconnected from those in the attached region. If we can recover that information, then it is possible to support a `detach` operation, which takes its argument and moves it into a newly-allocated region.

Typing rules for this proposed detach can be found in figure 4.33. We limit uses of `detach` to objects in regions reachable from some `isolated` field; when an object is `detached`, we effectively `retract` the `detach` target's region back into this `isolated` field. Unlike a normal `retract`, however, we preserve access to the target of the `detach`, placing it in its own newly-allocated region. For this operation to be sound, we must ensure that the object graph associated with the newly-detached object and the object graph reachable from the retracted `isolated` field are disjoint. Thus, the `detach` command also must have a runtime component: `detach` ensures dynamically that the object graph reachable from its target does not

intersect with the object graph reachable from the retracted field. If this check fails, the program panics.[2]

While this dynamic check is potentially expensive, we have identified two approaches which should significantly curtail runtime costs. The first option is to piggy back on the graph maintained by the garbage collector; enhancing the garbage collection routine to maintain state accessible to `detach` will hide much of the overhead of `detach`'s imposed graph traversals.

Should the garbage collection approach not prove viable, the second option is to manage `detach`'s complexity via a reference-counting strategy. In this strategy, each object will be equipped with a reference count tracking the number of *heap* references to that object (stack references are excluded). If an object's reference count is 0, then this means it can only be accessible directly via stack references, which will be invalidated statically after the `detach` and are thus not important in our reasoning. At the point of `detach`, a runtime mechanism walks the object graph reachable from the detach's target, excluding any `isolated` fields. It then compare the reference counts on the objects with the discovered graph. If the discovered graph fully accounts for all reference counts on these objects, and if the traversal does not encounter the to-be-retracted `isolated` field, then we can conclude that the detach target is indeed isolated from the surviving portions of its region.

Due to the possibility of runtime failure, we expect that programmers will only choose to employ `detach` in cases where they are sure the `detach`'s target is indeed isolated; this condition will likely keep the object graph that `detach` must traverse to a manageable size. We therefore

2 In a full language, this case would result in an exception instead, which may be handled.

hope that the runtime traversal check required by `detach` will in practice be efficient.

4.10 RELATED WORK

The type system we have defined in this chapter owes much to a rich history of language design over the past thirty years. In particular, this system mixes innovations from several strong lines of work:

- Ownership Types and capabilities
- Regions
- Linear types (and linear regions)

In this section we attempt to broadly characterize notable works from each school, and discuss the ways in which our system improves on them.

4.10.1 *Ownership Types and Nonlinear Uniqueness*

While we use the terminology of regions [Tofte and Talpin, 1994], *ownership contexts* from the ownership types literature [D. G. Clarke, Potter, and Noble, 1998] are actually our closer intellectual cousin. Originally proposed by Clarke, Potter, and Noble in 1998 [D. G. Clarke, Potter, and Noble, 1998], ownership types associate each object with an *ownership context*. Each field (or local declaration) may be annotated with a particular context, limiting the variables which may be assigned to this location to those owned by the declared context. These ownership contexts are quite similar to our use of *regions* in the type system presented in this chapter; at a high level,

the primary differences between these concepts is that ownership contexts are fixed; objects forever live within a single ownership context, and ownership contexts cannot be merged, consumed, or generated on the fly. This effectively means that objects can never change owner; patterns such as merging two collections are not possible under the original ownership work.

Recognizing these limitations, the PRFJ and AliasJava systems introduced the ability to mix ownership with uniqueness; many other languages subsequently followed their lead [Aldrich, Kostadinov, and Chambers, 2002; Boyapati and Rinard, 2001]. The idea of unique references without ownership information predates these systems; in object-oriented languages, their original popular incarnations were via the Eiffel*, Balloons, and Islands [Almeida, 1997; Hogg, 1991; Minsky, 1996]. Each language which uses uniqueness also introduces some notion of "borrowing" (similar to our `preserves` annotation on functions), which allows for the creation of temporary aliases to otherwise-unique objects. Beyond their borrowing exemptions, these languages all take uniqueness very literally: a unique reference is the only reference in existence which points to a particular object. Clarke and Wrigstad weakened this constraint by introducing the idea of *external* uniqueness, and with it the property of a *dominating reference*: an external-unique reference is traversed on all paths from roots to the objects to which it refers [D. Clarke and Wrigstad, 2003; D. Clarke, Wrigstad, et al., 2008]. These externally-unique references are quite similar to our [simple] `isolated` fields, with one important difference: our `isolated` fields dominate *all objects reachable* from their target, while external references need only dominate their target. This prevents externally-unique references from implying *transitive* ownership. In our

language, access to an `isolated` field implies unique access to all objects reachable from that field—a property missing from externally-unique references.

Variations on the ownership model exist. Owning object have been made explicit or abstracted via capabilities [J. Boyland, Noble, and Retert, 2001; Castegren and Wrigstad, 2016; Clebsch et al., 2015; Haller and Odersky, 2010]. A related line of work on "universes" views owners as modifiers [Müller and Poetzsch-Heffter, 1999]; this has been extended to thread safety [Cunningham, Drossopoulou, and Eisenbach, 2007]. Peter Müller and Arsenii Rudich extended universes with ownership transfer via a mechanism which bares superficial similarity to our focus/unfocus mechanisms [Müller and Rudich, 2007]; these are in fact unrelated (our focus mechanism comes from Vault [Fähndrich and DeLine, 2002]).

Throughout all this work on combining increasingly-sophisticated ownership types with refined ideas of uniqueness, one persistent crutch remains: a heavy reliance on `null`. Each of these systems has an equivalent of our language's `consumes` function, which renders the caller unable to re-use the argument after its execution. Unlike in our system, these works need to ensure that all (or at least some) references are valid at all times. Thus rather than make consumed arguments statically inaccessible, the vast majority of these systems employ a "destructive read" which implicitly nulls them instead [Aldrich, Kostadinov, and Chambers, 2002; Banerjee and Naumann, 2002; Boyapati, Lee, and Rinard, 2002; Boyapati and Rinard, 2001; D. Clarke, Wrigstad, et al., 2008], though other approaches (such as a "swap" primitive) do exist [Haller and Odersky, 2010; Jim et al., 2002]. Several systems adopt the technique of Alias Burying [J. Boyland, 2001] to avoid this implicit nulling when all possible aliases to a unique object

are "dead" (will never be used again); unlike our reference invalidation via region removal, alias burying cannot always eliminate all aliases and thus a reliance on implicit null still remains.

Relying on implicit null effectively transforms a static error (using an object which is outside the reservation) into a dynamic one (a null pointer exception). As we have argued throughout this chapter, managing reservations manually is error-prone; we must therefore expect that exceptions introduced to handle reservation misuse will commonly be thrown, requiring programmers to either fall back on the careful reasoning present in traditional languages, or to introduce exception handling code around every use of a unique reference—a major headache. Using a swap operation to avoid this null reliance is not a major improvement. This requires the programmer to always have a default value available when reading—an unrealistic requirement unless the object in question has a nullary constructor, which comes with similar challenges to pervasive null.

The reason these languages opt for a dynamic, rather than static, reference invalidation mechanism comes down to the difficulty of reasoning about *non-unique* aliasing. These systems all allow non-unique objects to contain unique fields; they also all do not precisely track non-unique objects. Thus they will in general be unable to determine if two particular non-unique references refer to the same object, and therefore also unable to determine if the unique fields within those objects refer to the same or different targets. These systems have effectively made a different choice in the trade-off space; they can always freely use unique fields of possibly-aliased non-unique objects, at the cost of ensuring that these unique fields are always valid. Our focus mechanism meanwhile blocks any access to `isolated` fields whose containing object may alias an already-tracked

object. This allows us to avoid relying on implicit null, or its close cousin the swap primitive.

4.10.2 *Linear Systems and Regions*

In linear type systems, the trade-off of introducing null vs. losing access to a potentially-large number of objects swings in entirely the opposite direction. This appears to be due to the languages which first played host to these ideas; while ownership was always intended for an object-oriented setting (in which null has long been a fact of life), linear and region types were originally introduced in a functional context (in which types cannot be assumed to have a nullary constructor) [Girard, 1987; Tofte and Talpin, 1994; Wadler, 1990]. The first popular toy language to hybridize linear and nonlinear references was Wadler's "Linear Types Can Change the World!" [Wadler, 1990], which while sound did not extend to settings with mutable data structures. Early work on linear type systems was characterized by a relatively rigid separation between linear and nonlinear objects; while a linear object may contain nonlinear data, the reverse is generally not possible [Odersky, 1992; Smetsers et al., 1994; Walker and Watkins, 2001]. Linear (and affine) type systems have since been adopted for many purposes including refinement types in Java [Degen, Thiemann, and Wehr, 2007], validating transactions, [Beckman, Bierhoff, and Aldrich, 2008], and most notably the correct use of protocols. Linear types for protocols has two main branches; the first is typestate for objects [DeLine and Fähndrich, 2004; Fähndrich and DeLine, 2002], and the second is session types [Vasconcelos, 2012], both of which have become fields of

study in their own right. The most popular embodiment of a traditional linear type system for memory management is Rust [Matsakis and Klock, 2014]; we compare with it directly in subsection 4.10.4, and allow it to stand as a proxy for traditional linear languages.

Tofte and Talpin introduced the idea of regions [Tofte and Talpin, 1994]. Initially, regions associated each allocation with a precise scope during which it was live; later, this was improved to the lifetime of a named *region* which could be explicitly allocated and deleted [Crary, Walker, and Morrisett, 1999], or even reasoned about explicitly [Henglein, Makholm, and Niss, 2001]. Regions enable safe stack-based memory management in a language with seemingly-dynamic allocation. The primary challenge of regions was closures, which might escape the lifetime of their closed-over variables' regions. To handle this, region-based typing introduced the idea of effect types on functions, explicitly listing the regions which it accessed [Tofte and Talpin, 1997]. The largest difference between regions and our system is that our regions aren't fixed. They can be merged, renamed, retracted into and explored out from other regions. This in turn removes the need for complex effect annotations on our functions; by retracting and merging regions, we can represent complex object graphs via their simple entry points. Classic regions, meanwhile, were built for memory management [Jim et al., 2002; Tofte, Birkedal, et al., 2001], and so must have a more precise view of the objects each region contains. More recently, this precision has been exploited to reason about precise object layouts for serialization [Vollmer et al., 2019].

Linear regions—regions whose lifetime is managed a la "Wadler-style" linear types—have long been a popular synergy. These hybrids introduce the ability to "open" a region and freely access the objects within it for a

limited scope [Fähndrich and DeLine, 2002; Walker and Watkins, 2001]. Several of the papers previously listed as explicitly linear or region-based in fact feature aspects of both [Beckman, Bierhoff, and Aldrich, 2008; DeLine and Fähndrich, 2004; Haller and Odersky, 2010]. The Cyclone project [Jim et al., 2002], having made several attempts to formalize their language [Fluet and Morrisett, 2006; Grossman et al., 2002; Hicks et al., 2003], also eventually settled on linear regions [Fluet, Morrisett, and Ahmed, 2006].

4.10.3 *Significant Complexity*

Several systems manage to succeed in ensuring isolation safety and avoiding implicit null (or swap), but in so doing overshoot our desired level of user-facing complexity [Boyapati and Rinard, 2001; J. T. Boyland and Retert, 2005; Castegren and Wrigstad, 2016; Clebsch et al., 2015; Jim et al., 2002]. Here the complexity does not appear to be incidental; it is not clear how to identify a "simple core" language which would be complete on its own. Indeed, my experiences designing the type system in this chapter speak to the speed at which complexity can creep in via even the most apparently-innocuous design choices.

4.10.4 *Closest Cousins and Famous Friends*

There are a few existing systems which come quite close to matching our language design goals; these are summarized in table 4.1. In this table, the "OwnerJ" row captures the close descendants of original ownership type systems, including PRFJ and AliasJava, as described in subsection

Table 4.1: A table of related work, indicating which of our "metrics for success" has been satisfied by existing systems. A check mark indicates a goal is satisfied; a cross indicates a goal is not satisfied; a tilde indicates that a feature is absent, but either could be added or is not applicable to that language's design.

Language	Abstraction	No null/swap	Arbitrary graphs	Simple
Rust	✓	✓	×	∼
Cyclone	∼	×	✓	×
CQual	×	✓	✓	✓
Unique	✓	∼	×	∼
Vault	✓	✓	∼	∼
Mezzo	∼	∼	∼	✓
Scala*	✓	×	✓	✓
OwnerJ	✓	×	✓	✓
Pony	✓	∼	∼	∼
M#	✓	×	✓	✓
Sing#	✓	×	×	✓

4.10.1. The "Unique" row captures the limitations of type systems in the style of Wadler's popularization [Wadler, 1990], described in subsection 4.10.2. This subsection proceeds with a detailed description of the projects in other rows of this table, and discusses how they relate to our work.

Rust. Much of the renewed interest in type systems suitable for reservation safety has focused on the emergence of Rust, the first such language to gain widespread adoption [Jespersen, Munksgaard, and Larsen, 2015; Jung et al., 2017; Matsakis and Klock, 2014; Reed, 2015; Weiss et al., 2019]. There are several essential differences between our system and Rust; the most fundamental of these is that Rust is among a number of languages which have difficulty forming graphs. This is due in a large part to Rust's notion of ownership and ownership as domination. In Rust, a field is owned by its encapsulating object. Also in Rust, a field cannot hold references to its

owner. This combination ensures that the pattern of a doubly-linked list we have used as our running example is difficult to implement in Rust; each node in the list is effectively a peer with the node it references, a concept which does not sit well with the Rust system. While it is possible to build object graphs out of borrowed references with a shared lifetime parameter, the process of doing so is quite complex—and still requires some associated owner object and associated borrow scope from which it cannot escape. These restrictions are similar to how our language would behave were we to have a single object per region; non-simple graphs are allowed, but the cost is a dramatic increase in static tracking, much of it borne directly by the user in the form of extra annotations.

Adoption and Focus. We owe our mechanism for focusing objects to Fähndrich and DeLine's Adoption and Focus from the Vault language [Fähndrich and DeLine, 2002]. While the ideas behind focus are largely the same in our two works, both the formal treatment and the setting in which they are deployed are less similar. Vault is a primarily linear language for reasoning about protocol state in which particular objects can be freely aliased, exempting them from the requirements of linearity. Unlike similar linear type systems up to this point, Vault's nonlinear objects are allowed to contain references to both linear or nonlinear objects. A linear field of a potentially nonlinear object in vault is roughly analogous to `isolated` fields in this type system. This analogy is rough, however; our `isolated` references may refer to objects which are freely aliased within their region, while Vault's linear fields must be unique references. As in our work, Vault prevents access[3] to `isolated` fields unless their containing object is *focused*.

3 though only for writing; reading is always permitted

As in our work, Vault needs to prevent two potential aliases from being focused simultaneously. To handle this, Vault introduces the notion of linear "guards" (analogous to our regions), and explicitly tracks "capabilities with keys" (analogous to our $\mathcal{H}$ environment) to determine which regions should be accessible for focus. Unlike our focus mechanism, Vault's focus is scoped; at the end of the scope, all linear fields of the focused objects must uniquely point to an object with typestate matching that of the declared field, but *during* the focus these fields may have their typestate changed or may be invalidated.

Vault's other operation, *adoption*, has no precise analogue in the type system presented here. Adoption in vault takes two linear objects, forms a reference from one to the other, and then returns the "owned" object via a guarded non-linear reference. The owner—the object which remains linear and contains a linear reference to this now-nonlinear object—effectively serves as the region in which this object lives. When the owner is destroyed, its internal reference to the owned object is made available, restoring the owned object's linear status (and necessitating that the owned object have no other aliases at the time). Guards, which we previously mentioned as analogues to regions, are effectively an abstraction of this owning object; this obviates the need to explicitly pass owners to functions which require the adopted object. This operation has elements of our `attach` construct, but actually bears more resemblance to plain assignment into `isolated` fields. Our choice to treat every object as freely aliased within a region avoids any need for the awkward step of "de-linearizing" an object accomplished via adoption, or the notion of explicit "owner" objects.

Crucially, Vault has no reasoning principle akin to the simple (or non-simple) regions/fields/objects in this type system; in Vault, what we would

view as violations of simplicity are permitted within focus blocks, but even here the linearity of "linear" fields must be maintained. Vault proposes an extension by which linear fields exposed via focus may themselves be adopted, resulting in the ability to alias a linear field within a focus scope; this mechanism is both unclear and appears remarkably difficult to use.

John Boyland lifted this adoption and focus mechanism directly to the domain of fractional permissions via a construct called *Nesting*, which additionally encompasses the idea of storing read-only aliases of a linear object within several different mutable aliasable objects, and leveraging the ideas of `focus` and `defocus`[4] to later recover unique access to them [J. T. Boyland, 2010].

In comparison, our system requires less rigid management of focused objects and does not enforce linearity on `isolated` objects themselves—just on their regions. In our system, all references—even simple `isolated` references—can point to objects which participate in cycles; this would not be possible in Vault, reducing the ease with which Vault can implement the running doubly-linked list example.

To the best of our knowledge, there is only one other mechanism which bears similarity to focus: the `restrict` [Foster, Terauchi, and Aiken, 2002] and `confine` [Aiken et al., 2003] qualifiers from CQual. In this line of work, the authors propose a monomorphic type system which leverages a flow-insensitive alias analysis to generate a set of abstract objects, which are akin to the regions in this work or the "owners/adopters" in Vault. Like other region work of this era, CQual's type system then uses effect types to determine which abstract objects may be accessed in a given scope. The `restrict` and `confine` constructs generate a scope in which a single

4 Boyland dropped the "focus must have a scope" requirements of Vault

variable (or pure expression in the case of `confine`) is elevated to the sole accessible reference for some abstract object, allowing strong updates to occur on that object. This is analogous to `focus` introducing the ability to read and modify `isolated` fields. While CQual's mechanism is similar, certain sensible design choices it makes for the domain of low-level C programs are not a match for our domain; in particular, it relies on the ability to perform a global inter-procedural analysis.

Capabilities in Scala. In their 2010 paper [Haller and Odersky, 2010] Philipp Haller and Martin Odersky set out to design a language with nearly identical goals to those we outlined here. Though derived independently, our surface features are eerily similar; both our work supports an idea of `isolated` (`@unique`) fields which dominate their entire reachable object graph, and annotate methods with `preserves` (`@transient, @peer`) or `consumes` (`@unique` on methods). Both languages back these surface features with static labeled regions (or capabilities), which are consumed whenever an object within the region (guarded by the capability) is rendered invalid. We both even share the `attach` (`capture`) virtual command. This is where the similarity ends, however; the system proposed by Haller and Odersky has no equivalent to `focus` or `explore`, and can only read `isolated` (`@unique`) fields by `swapping` them with an object in some disjoint region. This is only slightly better than relying on implicit `null`; it assumes that every context in which one wishes to dereference an `isolated` field contains the means to construct a replacement for it. In practice, this will likely result in the programmer opting for explicit option types, and manually swapping with a `None` value.

Sing#, M# and Pony. Sing# is a language developed at Microsoft Research to enable the single-process-space operating system Singularity

[Fähndrich et al., 2006]. Sing# includes a basic notion of region types via their *placed types*, which reside in a special region of memory and can be shared among processes. At some point near 2008, the Singularity project moved from Microsoft Research and evolved into the aimed-at-production Midori project, which came with its own language, M#.[5] This language is far more fully-featured, and has a notion of `isolated` types quite like the `isolated` fields presented in this chapter [Gordon et al., 2012]. It also includes a notion of a "recovery" scope, which allows a program to temporarily violate isolation. It does not however reason about aliasing at all, relying on either immutability or destructive reads to access `isolated` fields of non-`isolated` objects, exposing programmers once again to the dangers of implicit null. Pony, a more recent language out of Microsoft research, uses a similar set of keywords and a similar notion of a recovery scope, but it too does not reason precisely enough about aliasing to support `focus`-style reads [Clebsch et al., 2015]. Unlike many of the languages discussed in this section, these languages' target domain is safe concurrent code in which isolation is the primary reasoning principle.

Mezzo. Mezzo is a recent language in the ML family featuring duplicable and affine permissions in a larger language. Mezzo integrates many of the mechanisms of work that has come before; it incorporates a form of fractional permissions to allow immutability, it easily builds immutable graphs, and it leverages adoption (inspired by but not the same as [Fähndrich and DeLine, 2002]) to allow the formation of non-tree object graphs. Mezzo's treatment of adoption includes one essential feature not found in other related work—the ability to destructively read a potentially-aliased

5 Oddly enough the best source of this I have is a long-since-deleted job post once located here: https://careers.microsoft.com/jobdetails.aspx?ss=&pg=0&so=&rw=11&jid=111551&jlang=EN&pp=SS

field of an object. This object is then retrieved into its own region; subsequent attempts to access it via its original region will cause a runtime error. This version of destructive reading is similar to the `detach` construct in this chapter, except that `detach` is built to work in harmony with our `focus` mechanism, and thus must fail at the point of detaching if the detaching object graph is still reachable (or can still reach) objects which remain in the original region. Mezzo's novel destructive read, meanwhile, allows references to exist within the region and makes it a dynamic error to use them. While Mezzo does indicate that extensions involving `focus` are possible, it did not pursue them and has not determined how such a mechanism would interact safely with destructive reads. It is also unclear if Mezzo's adoption mechanism allows the formation of arbitrary graphs, or only DAGs. This combination of limitations makes it difficult to see how a doubly-linked list could be implemented in Mezzo without relying on Mezzo's version of implicit null.

4.11 CONCLUSION

By leveraging the idea of simplicity, we have built a type system which uses regions to ensure reservation safety, *without* introducing uninferrable user-facing complexity, relying on null, or forcing the entire object graph to form a forest. This type system is an appropriate match for Gallifrey's needs, and can be used to ensure restrictions in Gallifrey are respected.

5

CONCLUSION

Writing programs against weak consistency does not always doom the programmer to an endless parade of hard-to-find bugs. Strong consistency *can* scale, especially within a single datacenter. The story which opened this dissertation—the story of a trade-off between correctness on the one hand and performance on the other—is increasingly becoming the story of yesterday. Through the work presented in this dissertation and the work of countless others, we are pushing the domain of consistency past the familiar abstraction of reads and writes, and into the application domain. We are reasoning about the consistency of the *whole application*, not the consistency of some intermediate layer (e.g. memory) within it.

5.1 MIXT

The first tentative steps in this direction come with MixT, presented in chapter 1. MixT argues that taking a purely operational view of consistency is short-sighted: that the consistency of operations is derived from data invaraints, and thus the better place to reason about consistency is at the level of individual objects, not individual operations. MixT leverages this idea to introduce *mixed-consistency transactions*, which can manipulate information with multiple consistency levels in a single transaction. MixT's key technology is an adaptation of a traditional integrity information-flow

type system, used within individual transactions to ensure that weakly-consistent observations cannot unduly influence strongly-consistent mutations. MixT's type system guarantees that all well-typed mixed-consistency transactions will have a strict separation between more-consistent and less-consistent information; while more-consistent information can influence less-consistent information, the reverse is impossible without endorsement. MixT takes advantage of this separation to implement mixed-consistency transactions via a novel consistency-aware transaction splitting translation, compiling each mixed-consistency transaction down to a sequence of linked single-consistency transactions.

5.2 MONOTONICITY

While MixT recognizes that consistency is best considered at the level of information rather than operations, it still has a rather traditional take on the role of weak consistency: that while weak consistency invites unintuitive errors, its use is necessary to avoid expensive synchronization. As demonstrated by Derecho and Gallifrey, this doesn't have to be true. The key insight at the heart of both Derecho and Gallifrey is monotonicity. Data that is shared monotonically only grows; mutations to monotonic data must inflate it according to some order, and observations of monotonic data can only determine lower bounds in this order. By using monotonicity, programs can be consistent—and convergent—by construction, even if the underlying replication is weakly consistent. Put simply, monotonicity is one way to decouple the consistency of an *application* from the consistency of the storage upon which it is built. This idea of leveraging monotonicity

to write convergent programs has been explored previously [Conway et al., 2012; Kuper and Newton, 2013; Meiklejohn and Van Roy, 2015]; where Derecho and Gallifrey innovate is in how this basic abstraction is exposed to the programmer.

5.3 DERECHO

In Derecho (chapter 2), monotonicity is confined to a core language over a Shared State Table (SST) synchronized via Remote Direct Memory Access (RDMA). This SST provides the abstraction of a vector of monotonic single-writer registers, storing booleans, integers, or small sets. Programs atop these registers are composed of *reducers*—for example min, max, or set union—which fold monotonic operators across the entire vector, producing a summary of its contents as a single value. While the SST's core language is limited, it is expressive enough to implement consensus with minimal synchronization, unlocking impressive performance for state machine replication. Though Derecho does allow programmers to interact directly with the monotonic SST, its primary user-facing abstraction is the strongly-consistent replicated object. Using Derecho's replicated objects, programmers can build services comprised of constellations of actors which communicate via a custom remote method invocation (RMI) framework. Our evaluation shows that users of these replicated objects enjoy performance benefits exceeding an order of magnitude greater than would be possible with the standard tools in this space.

5.4 GALLIFREY

Gallifrey (chapter 3) unlocks the promise of monotonic programming at the core of Derecho and generalizes it, allowing programmers to share any object by restricting its interface to only those operations which are safe to call concurrently. Individual objects may support more than one valid restriction; for example, a boxed integer could be safely shared under a restriction which allows incrementing and testing if the counter is *at least* a certain value, or under a restriction which allows decrementing and testing if the counter is *at most* a certain value. Gallifrey also allows objects to transition between different restrictions, changing the allowed operations on a single object over time. For example, an election object could initially be shared under a restriction that allows voting, and subsequently be shared under a restriction which allows tallying of votes.

5.5 A TYPE SYSTEM FOR GALLIFREY

Key to Gallifrey's promise is its ability to seamlessly integrate the world of monotonic shared objects with the world of unrestricted local objects. This ability is enabled by a novel linear, region-based type system (presented in chapter 4) which enforces isolation between shared objects In particular, this type system enforces that access to any object guarded by a restriction may only be made via that restriction. Chapter 4's type system achieves this with a minimum of user-facing complexity, while preserving the familiar abstractions of Java, permitting arbitrarily-shaped object graphs, and avoiding any reliance on null references.

5.6 FUTURE DIRECTIONS

The results of this dissertation present opportunities for extension in all directions. MixT's use of an information-flow type system leads to questions about the formulation of consistency itself—presenting an opportunity to reformulate consistency models not as a simple trace property for storage layers, but as a deeper semantic condition on applications in general. Derecho's use of a simple monotonic table to build consistent replication demonstrates that monotonicity is applicable in more situations than were previously explored. This calls into question our understanding of monotonic logic programming; what are the true limits of monotonicity? And when a protocol could be phrased either monotonically or non-monotonically, what principles should be employed in making that choice? Gallifrey—and its enabling type system—open up a world of possibilities in both the language and system design space; the future directions outlined in section 3.9.1 only begin to scratch the surface of these.

5.7 WRAPPING UP

Taken together, the works presented in this dissertation constitute an argument that the consistency of replication need not always constrain the consistency of an application. With MixT, Derecho, and Gallifrey, programmers can write correct programs atop weak consistency—by isolating weak components from strong, layering consistent replication atop weak consistency, or restricting operations to those guaranteed to be correct under weak consistency. This technology improves upon the state of the

art in distributed programming, enabling programmers to enjoy both the speed of weakly-consistent replication and the clear semantics of strong consistency.

APPENDICES

A

APPENDIX 1

This appendix contains sample programs for Gallifrey, capturing both the shared references introduced in chapter 3 and the type system introduced in chapter 4. These sample programs rely on the full design of Gallifrey and chapter 4's type system, including elements not yet supported by the Gallifrey compiler.

A.1 A DOUBLY-LINKED LIST

```
class List<T> {
  static class ListNode {
    ListNode next;
    ListNode prev;
    isolated T payload;

    public ListNode(consumes T payload){
      this.next = this;
      this.prev = this;
      this.payload = payload;
    }

    public ListNode(consumes (ListNode next, ListNode prev),consumes T
        payload ){
      this.next = next;
      this.prev = prev;
      this.payload = payload;
    }
  }

  isolated ListNode head;

  void add_to_list(consumes T item){
    ListNode old_head = this.head;
    this.head = new ListNode(old_head, old_head.prev, item);
    old_head.prev = this.head;
  }
```

```
  isolated T remove_item(
  preserves (preserves T -> isolated bool) which){
    ListNode curr = this.head;
    while (!which(curr.payload)){
       curr = curr.next;
    }
    curr.prev.next = curr.next;
    curr.next.prev = curr.prev;
    curr.next = curr;
    curr.prev = curr;
    ListNode ret = detach(curr);
    return ret.payload;
  }

  void for_each(preserves U foldarg, preserves (preserves U, preserves T
      -> void) foldfun){
    curr = this.head;
    do {
      foldfun(foldarg,curr.payload);
      curr = curr.next;
    } while (curr != l.head);
  }

  void map(preserves: consumes T -> isolated T mapfun){
    curr = this.head;
    do {
      curr.payload = mapfun(curr.payload);
      curr = curr.next;
    } while (curr != l.head);
  }
}
```

A.2 SIMPLE CHANNELS

```
interface Queue<T>{
   void enqueue_item(T t);
   isolated T dequeue_item() throws NoItemException;
   bool item_ready();
};

Restriction EnqueueOnly for Queue {
   allows enqueue_item;
};

Restriction SingleDequeuer extends EnqueueOnly for Queue {
   allows as test item_ready;
   allows dequeue_item contingent AlreadyDequeued;
};

abstract class Channel<Message>{
   shared Restriction[SingleDequeuer] Queue<Message> event_queue = new
       Queue<Message>();
   abstract void run_action(Message m);

   public void process_messages(){
      while (true){
         when (event_queue.item_ready()){
            Branch dequeue_branch = branch(event_queue){
```

```
            T item = event_queue.dequeue_item();
          };
          dequeue_branch.commit();
          //This branch is committed, so peek isn't returning anything
              provisional anymore.
          //it may be a stale value, but it's a "valid" stale value.
          run_action(dequeue_branch.peek[item]);
        }
      }
    }
    public shared Restriction[EnqueueOnly] Queue<Message>
        get_submission_queue(){
      return event_queue;
    }
}
```

A.3 COUNTER

```
class Counter {
    int count = 0;
    public void increment(){
        ++count;
    }
    public int get(){
        return count;
    }

    public bool at_least_target(int threshold){
        return count >= threshold;
    }
};

Restriction AddMostly for Counter {
    allows increment;
    allows as test at_least_target;
    allows get contingent MissedSomeIncrements;
    merge (increment i, get g -> int count){
      reject g with MissedSomeIncrements();
    }
}



Restriction DisplayMostly for Counter {
    allows get;
    allows at_least_target;
    allows increment contingent StateDisplayed;
    merge (increment i, get g -> int count){
      reject i with StateDisplayed();
    }
}

Restriction AddOrDisplay = AddMostly | DisplayMostly;
```

A.4 ELECTION

```
class Candidate {
   public final String name;
   isolated shared[CountOrDisplay] Counter c;
   Branch observing;

   public void vote() throws InconsistentVote{
      match c with
      | shared[AddMostly] c ⇒ c.increment();
      | shared[DisplayMostly] _c ⇒ {
         pull(); //if in a branch, synchronize with external events
         candidateBranch = branch(_c){_c.increment};
         candidateBranch.peek[_c].endorse(StateDisplayed(i) ⇒ throw new
             InconsistentVote(););
         candidateBranch.commit();
      }
   }

   provisional int estimate_count() {
      match c with
      | shared[DisplayMostly] c ⇒ return c.get();
      | shared[AddMostly] _c ⇒ return observing.open(_c){i = _c.get();}.
          peek[i];
   }
};
```

A.5 VERSION CONTROL

```
class Repository {
   String name;
   shared[AddOnly] Set<Commit> commit_list;
   shared[AddOnly] Set<Head> head_list;
   Branch[shared[AddOnly] Set<Commit> commit_list,
        shared[AddOnly] Set<Head> head_list,
        shared[RO] Commit new_commit,
        pc : InvalidatableRead] working_branch;
   local Head current_head;

   public void init() {
      //init
      working_branch = branch(commit_list, head_list, current_head){
         local Commit new_commit = new Commit("current",
                              current_head.current_commit,
                              current_head.current_commit.file_list);
      };
   }

   public void add(consumes File f) {
       working_branch.open(f) {
          new_commit.add_file(f);
       };
   }
```

```
    public void commit(preserves String message) {
        working_branch(message){
            new_commit.set_name(message);
            commit_list.add(new_commit);
            current_head.current_commit = new_commit;

        };
        working_branch.commit();
        init();
    }

    public void log() {
        local Commit commit = current_head.current_commit;
        while (commit != null) {
            System.out.println(commit.name);
            commit = commit.parent;
        }
    }

    public void checkout(consumes Head h) {
        head_list.add(current_head);
        current_head = h
        working_branch.abort();
        init()
    }

    void pull(preserves Head h) {
        working_branch.open(h) {
            // prompt the user to manually merge files
        }
    }
};

class Commit {
    String name;
    isolated Commit parent;
    isolated Set<File> file_list;

    Commit(preserves String commit_name, consumes Commit parent_commit,
         consumes Set<File> files){
        set_name(commit_name);
        parent = parent_commit
        file_list = files
    }

    void set_name(preserves String nm) {
        name = nm
    }

    void add_file(consumes File f) {
        //Remove file of the same name to be replaced
        file_list.insert(f);
    }

    void remove_file(consumes File f) {
        file_list.erase(f);
    }
};

class Head {
    String name;
    isolated Commit current_commit;

    Head(preserves String nm, consumes Commit commit) {
```

```
      name = nm;
      current_commit = commit;
   }
};

class File {
   String name;
   isolated List<String> lines;
};
```

A.6 ACTORS

```
class Channel<Send, Rcv>{
  private shared[appendOnly] Queue<Send,Rcv> requests;
  private shared[dequeProvisional] Queue<Rcv,Send> responses;

  public void send(consumes Send m){
    requests.enq(m);
  }

  public isolated Rcv rcv() {
    when (responses.non_empty()){
      Branch b = branch(responses){
        ret = responses.deq();
      }
      //if there is only one replica of this Queue with
      //access to provisional deq, then this commit should be "Free"
      b.commit();
      return b.peek[ret];
    }
  }
}


interface ChannelProducer<Send,Rcv>{
  public isolated Channel<Send,Rcv> connect();
}


class ChannelManager<Send,Rcv> implements ChannelProducer<Send,Rcv>{
  public shared[EnqMostly] Queue<Channel<Rcv,Send>> created;
  public Channel<Send,Rcv> connect(){
    shared[appendMostly] Queue<Send,Rcv> dir_1 = new Queue<Send,Rcv>();
    shared[appendMostly] Queue<Rcv,Send> dir_2 = new Queue<Rcv,Send>();
    //the more-restricted restrictions channel's constructor
    //takes are supertypes of this restriction
    created.insert(new Channel(dir_2,dir_1));
    return new Channel(dir_1,dir_2,);
  }
}

Restriction SingleUser for ChannelManager<Send,Rcv> {
  allows connect();
  //because created.deq is provisional, so must this be
  allows created.deq() contingent AlreadyDequeued;
  allows test created.non_empty();
};

Restriction All for ChannelProducer<Send,Rcv>{
```

```
  allows connect();
};

abstract class Server<Send,Rcv>{
  shared[SingleUser] ChannelManager<Send,Rcv> new_queues;
  isolated List<Channel<Rcv,Send>> queues;
  public abstract void handle_message(consumes Send s, preserves Channel<
      Rcv,Send> chan);
  public void receive(){
    while (true){
      when any {
        | new_queues.non_empty() =>
        branch(new_queues, queues){queues.enq(new_queues.created.deq());}.
            commit();
        | queues.fold(false, λ q, a . q.responses.non_empty() || a) =>
        handle_message(queues.filter(λ q . q.responses.non_empty()).first()
            .rcv());
      }
    }
  }
}
```

A.7 SHOPPING CARTS

A.7.1 *Centralized Shopping Cart*

```
class ShoppingCart{...}
Restriction Shopping for ShoppingCart{...}
Restriction Checkout for ShoppingCart{...}
Restriction All for ShoppingCart = Shopping | Checkout;

using CartServer_i = Server<SessionToken,shared[All] ShoppingCart>;
using CartConnectorService = ChannelManager<SessionToken,shared[All]
    ShoppingCart>;
using CartConnector = ChannelProducer<SessionToken,shared[All]
    ShoppingCart>;

class SingleCartServer extends CartServer_i{

  public SingleCartServer(){
    shared[SingleUser] CartConnectorService cm = new CartConnectorService()
        ;
    //this is an error if run more than once, so this server is a
        singleton.
    gal://localhost/CartConnector/All/getCart = cm;
    super(cm);
  }

  public void receive(){
    super.receive();
  }

  isolated map<SessionToken, shared[All] ShoppingCart> carts;
  public void handle_message(consumes SessionToken req_tok, preserves
      Channel<shared[All] ShoppingCart, SessionToken> chan){
    chan.send(carts[req_tok]);
```

```
  }
}


class CartClient {
  Channel<SessionToken, shared[All] ShoppingCart> cart_channel = gal://
      localhost/CartConnector/All/getCart.connect();
  shared[All] ShoppingCart my_cart;

  CartClient(){
    cart_channel.send(SessionToken);
    my_cart = cart_channel.receive();
  }

  /*
    other shopping-related code goes here...
   */
}
```

A.7.2 *Distributed Shopping Cart*

```
class ShoppingCart{...}
Restriction Shopping for ShoppingCart{...}
Restriction Checkout for ShoppingCart{...}
Restriction SthenC = Shopping | Checkout;


class ShoppingCartService{
  public shared[SthenC] ShoppingCart new_cart();
  public void checkout();
};

class ShoppingCartServer{
  public shared[SthenC] ShoppingCart new_cart(){
    shared[SthenC] ShoppingCart sc = new ShoppingCart();
    return sc;
  }


  public void checkout(shared[SthenC::Checkout] ShoppingCart done_cart){
    /* does some business logic */
  }

};

//somewhere in the shopping cart server...

main(){
  localhost/shared[All] ShoppingCartService cart_service = new
      ShoppingCartServer();
}
```

B

APPENDIX 2

In this appendix, we present the type system from chapter 4 with all the rules an syntax in one place.

B.1 SYNTAX

B.1.1 *Surface Syntax*

$$
\begin{aligned}
p &::= e \mid [\mathcal{H}]\ \ell\ \tau\ \mathit{fname}[H](\Gamma)\{e\}; p \\
&\quad \mid q_{re}\ \tau\ \mathit{fname}(q(\tau\ x, \ldots), \ldots); p \\
e &::= v \mid x \mid x = e \mid e.f \mid e.f = e \mid e \oplus e \mid e; e \mid e(e) \mid \mathit{fname} \\
&\quad \mid \texttt{if}\ (e)\ \{e\}\ \texttt{else}\ \{e\} \mid \texttt{while}\ (e)\ \{e\} \\
&\quad \mid \texttt{declare}\ x : \tau\ \texttt{in}\ \{e\} \mid \texttt{focus}\ x \mid \texttt{explore}\ x.f \\
&\quad \mid \texttt{retract}\ e \mid \texttt{unfocus}\ x \mid \texttt{attach}\ e\ \texttt{to}\ e \\
&\quad \mid \texttt{send-}\tau \mid \texttt{receive-}\tau \\
v &::= n \mid \texttt{true} \mid \texttt{false} \mid l \mid \mathit{constant} \\
\oplus &::= ; \mid + \mid * \mid - \mid \texttt{\&\&} \mid \texttt{||} \\
\mathit{fields}(Cls) &::= \{f : q_r\ \tau, \ldots\} \\
q_r &::= \texttt{isolated} \mid \cdot \\
\tau &::= Cls \mid \ \texttt{int} \mid \texttt{bool} \mid \texttt{void} \mid (q\ \tau, \ldots \to q_{re}\ \tau) \\
q &::= \texttt{consumes} \mid \texttt{preserves} \\
q_{re} &::= \texttt{isolated} \mid k
\end{aligned}
$$

B.1.2 *Typing Contexts and Virtual Commands*

$$\Gamma ::= x : \ell\ \tau, \Gamma \mid \cdot$$

$$\mathcal{H} ::= \ell\langle\rangle, \mathcal{H} \mid \ell\langle X\rangle, \mathcal{H} \mid \cdot$$

$$\mathrm{P} ::= l : \ell\ \tau, \mathrm{P} \mid \cdot$$

$$X ::= x[F], X \mid \cdot$$

$$F ::= f \mapsto \ell, F \mid \cdot$$

$$cmd ::= \texttt{focus}\ \ell@x \mid \texttt{explore}\ \ell@x.f \mid \texttt{retract}\ \ell \mid \texttt{unfocus}\ \ell@x$$
$$\mid \texttt{attach}\ \ell\ \texttt{to}\ \ell \mid \texttt{consumes}\ \ell \mid \texttt{preserves}\ \ell$$

B.2 TYPING RULES

B.2.1 *Structures, Assignment, Sequence*

VARREF

$$\frac{\mathcal{H} \vdash \ell}{\mathcal{H}; \Gamma, x : \ell\ \tau; \mathrm{P} \vdash x : \ell\ \tau \dashv \mathcal{H}; \Gamma}$$

SEQUENCE

$$\frac{\mathcal{H}; \Gamma; \mathrm{P} \vdash e_1 : \ell_1\ \tau_1 \dashv \mathcal{H}'; \Gamma' \qquad \mathcal{H}'; \Gamma'; \mathrm{P} \vdash e_2 : \ell_2\ \tau_2 \dashv \mathcal{H}''; \Gamma''}{\mathcal{H}; \Gamma; \mathrm{P} \vdash e_1 ; e_2 : \ell_2\ \tau_2 \dashv \mathcal{H}''; \Gamma''}$$

FIELD REFERENCE

$$\frac{\mathcal{H}; \Gamma; \mathrm{P} \vdash e : \ell\ \tau_0 \dashv \mathcal{H}'; \Gamma' \qquad f : q_r\ \tau \in \mathit{fields}(\tau_0) \qquad \mathcal{H}' \vdash q_r\ e@\ell.f : \ell' \qquad \mathcal{H}' \vdash \ell'}{\mathcal{H}; \Gamma; \mathrm{P} \vdash e.f : \ell'\ \tau \dashv \mathcal{H}'; \Gamma'}$$

FIELD ASSIGNMENT

$$\frac{\mathcal{H}; \Gamma; \mathrm{P} \vdash e_1 : \ell\ \tau_0 \dashv \mathcal{H}'; \Gamma' \qquad f : q_r\ \tau \in \mathit{fields}(\tau_0) \qquad \mathcal{H}'; \Gamma'; \mathrm{P} \vdash e_2 : \ell'\ \tau \dashv \mathcal{H}''; \Gamma'' \qquad \vdash q_r\ e_1@\ell.f = \ell' : \mathcal{H}'' \Rightarrow \mathcal{H}'''}{\mathcal{H}; \Gamma; \mathrm{P} \vdash e_1.f = e_2 : \ell'\ \tau \dashv \mathcal{H}'''; \Gamma'}$$

ASSIGN-Γ

$$\frac{\mathcal{H}; \Gamma; \mathrm{P} \vdash e : \ell\tau \dashv \mathcal{H}'; \Gamma', x : \ell_i\tau \qquad \mathcal{H}' \vdash x@\ell_i = \ell}{\mathcal{H}; x : \ell_0\tau, \Gamma; \mathrm{P} \vdash x = e : \ell\ \tau \dashv \mathcal{H}'; \Gamma', x : \ell\tau}$$

B.2.2 *Functions*

APPLY

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash e_f : \ell_f\ (q\ \tau \to q_r\ \tau') \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma';\mathrm{P} \vdash e : \ell_e\tau \dashv \mathcal{H}'';\Gamma'' \qquad \Gamma'' \vdash (q\ \ell_e \to q_r\ \ell) : \mathcal{H}''' \Rightarrow \mathcal{H}_{out}}{\mathcal{H};\Gamma;\mathrm{P} \vdash e_f(e) : \ell\tau \dashv \mathcal{H}_{out};\Gamma''}$$

LOOKUP

$$\frac{(\mathit{fname} : \tau) \in \mathcal{F} \qquad \ell\ \mathit{fresh}}{\mathcal{H};\Gamma;\mathrm{P} \vdash \mathit{fname} : \ell\ \tau \dashv \Gamma;\mathcal{H}, \ell\langle\ \rangle}$$

DEFINE

$$\frac{\mathit{fname} : (q\ \tau \to q_r\ \tau') \in \mathcal{F} \qquad \ell\langle\ \rangle; x : \ell\ \tau; \cdot \vdash e : \ell'\ \tau' \dashv \mathcal{H}_{f1}, \mathcal{H}_{f_2}; \Gamma'_f \qquad \cdot \vdash (q\ \ell \to q_r\ \ell'') : \ell\langle\ \rangle \Rightarrow \mathcal{H}_{f1}}{\vdash q_r\ \tau'\ \mathit{fname}(q\ \tau\ x)\{e\}}$$

B.2.3 *IMP*

CONSTANT

$$\frac{\ell \textit{ fresh}}{\mathcal{H};\Gamma;\mathrm{P} \vdash n : \ell\ \texttt{int} \dashv \mathcal{H}, \ell\langle\,\rangle;\Gamma}$$

BOOL-CONSTANT

$$\frac{b \in \{\texttt{true}, \texttt{false}\} \qquad \ell \textit{ fresh}}{\mathcal{H};\Gamma;\mathrm{P} \vdash b : \ell\ \texttt{bool} \dashv \mathcal{H}, \ell\langle\,\rangle;\Gamma}$$

DECLARE

$$\frac{x \notin \Gamma \qquad x \notin \mathcal{F} \qquad \mathcal{H};\Gamma, x : \bot\ \tau;\mathrm{P} \vdash e : \ell\ \tau' \dashv \mathcal{H}';x : \ell'\tau,\Gamma'}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{declare}\ x : \tau\ \texttt{in}\{e\} : \ell\ \tau' \dashv \mathcal{H}';\Gamma'}$$

INFIX

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash e_1 : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma';\mathrm{P} \vdash e_2 : \ell\ \tau \dashv \mathcal{H}'';\Gamma'' \qquad \vdash \tau \oplus \tau}{\mathcal{H};\Gamma;\mathrm{P} \vdash e_1 \oplus e_2 : \ell\ \tau \dashv \mathcal{H}'';\Gamma''}$$

CONDITIONAL

$$\frac{\begin{array}{c}\mathcal{H};\Gamma;\mathrm{P} \vdash e_1 : \ell_b\ \texttt{bool} \dashv \mathcal{H}';\Gamma' \\ \mathcal{H}';\Gamma';\mathrm{P} \vdash e_2 : \ell_l\tau \dashv \mathcal{H}_l, \mathcal{H}_2;\Gamma_l,\Gamma_2 \qquad \mathcal{H}';\Gamma';\mathrm{P} \vdash e_3 : \ell_r\tau \dashv \mathcal{H}_r, \mathcal{H}_3;\Gamma_r,\Gamma_3 \\ \mathcal{H}_l;\Gamma_l \equiv_m \mathcal{H}_r;\Gamma_r \qquad m(\ell_l) = \ell_r \qquad \mathit{vars}(\Gamma) = \mathit{vars}(\Gamma_l)\end{array}}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{if}(e_1)\ \{e_2\}\ \texttt{else}\ \{e_3\} : \ell_l\ \tau \dashv \mathcal{H}_l;\Gamma_l}$$

WHILE

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash e_1 : \ell_b\ \texttt{bool} \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma';\mathrm{P} \vdash e_2 : \ell\ \tau \dashv \mathcal{H}_l, \mathcal{H}_2;\Gamma_l,\Gamma_2 \qquad \mathcal{H}';\Gamma' \equiv_m \mathcal{H}_l;\Gamma_l}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{while}(e_1)\ \{e_2\} : \bot\ \texttt{void} \dashv \mathcal{H}';\Gamma'}$$

B.2.4 *Virtual Commands*

FOCUS

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash x : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{focus}\ \ell@x \dashv H' \Rightarrow \mathcal{H}''}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{focus}\ x : \ell\ \tau \dashv \mathcal{H}'';\Gamma'}$$

EXPLORE

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash x : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{explore}\ \ell@x.f : \mathcal{H}' \Rightarrow \mathcal{H}'' \qquad \mathcal{H}'';\Gamma';\mathrm{P} \vdash x.f : \ell'\ \tau \dashv \mathcal{H}''';\Gamma''}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{explore}\ x.f : \ell' \dashv \mathcal{H}''';\Gamma''}$$

RETRACT

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash e : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{retract}\ \ell : \mathcal{H}' \Rightarrow \mathcal{H}''}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{retract}\ e : \bot\ \texttt{void} \dashv \mathcal{H}'';\Gamma'}$$

UNFOCUS

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash x : \ell\ \tau \dashv \mathcal{H}';\Gamma' \qquad \Gamma' \vdash \texttt{unfocus}\ \ell@x : \mathcal{H}' \Rightarrow \mathcal{H}''}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{unfocus}\ x : \ell\ \tau \dashv \mathcal{H}'';\Gamma'}$$

ATTACH

$$\frac{\mathcal{H};\Gamma;\mathrm{P} \vdash e_1 : \ell_1\ \tau_1 \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma';\mathrm{P} \vdash e_2 : \ell_2\ \tau_2 \dashv \mathcal{H}'';\Gamma'' \qquad \Gamma'' \vdash \texttt{attach}\ \ell_1\ \texttt{to}\ \ell_2 : \mathcal{H}'' \Rightarrow \mathcal{H}'''}{\mathcal{H};\Gamma;\mathrm{P} \vdash \texttt{attach}\ e_1\ \texttt{to}\ e_2 : \ell_2\ \tau_1 \dashv \mathcal{H}''';\Gamma''[\ell_1 \mapsto \ell_2]}$$

B.2.5 *Locations and Communication*

LOCATION-REFERENCE

$$\mathrm{P}, l : \ell\ \tau \vdash l : \ell\ \tau$$

LOCATIONS

$$\frac{\mathrm{P} \vdash l : \ell\ \tau \qquad \mathcal{H} \vdash \ell}{\mathcal{H};\Gamma;\mathrm{P} \vdash l : \ell\ \tau \dashv \mathcal{H};\Gamma}$$

$$\mathcal{H}; \Gamma; \mathrm{P} \vdash \texttt{send-}\tau : (\texttt{consumes}\ \tau \to \texttt{void}) \dashv \Gamma; \mathcal{H}$$

$$\mathcal{H}; \Gamma; \mathrm{P} \vdash \texttt{receive-}\tau : (\texttt{void} \to \texttt{isolated}\ \tau) \dashv \Gamma; \mathcal{H}$$

B.3 HEAP RULES

B.3.1 *Well-Formed*

$\Gamma;\mathcal{H}$ WELL-FORMED

$$\frac{\forall x : \ell\ \tau \in \Gamma,\ \ell'\langle x[\ldots], \ldots\rangle \in \mathcal{H} \colon \ell = \ell'}{\vdash \Gamma; \mathcal{H}}$$

B.3.2 *Fields and Assignment*

ISOLATED-FIELD-REFERENCE

$$\mathcal{H}', \ell\langle x[F, f \mapsto \ell']\rangle \vdash \texttt{isolated}\ x.f@\ell : \ell'$$

ISOLATED-FIELD-ASSIGNMENT

$$\vdash \texttt{isolated}\ x.f@\ell = \ell' : \mathcal{H}, \ell\langle x[F, f \mapsto \ell'']\rangle \Rightarrow \mathcal{H}, \ell\langle x[F, f \mapsto \ell']\rangle$$

NON-ISOLATED FIELDS

$$\mathcal{H} \vdash \cdot\ e.f@\ell : \ell$$

NON-ISOLATED ASSIGNMENT

$$\vdash \cdot\ e.f@\ell = \ell : \mathcal{H} \Rightarrow \mathcal{H}$$

REGION-VALID

$$\mathcal{H}, \ell\langle X \rangle \vdash \ell$$

UNTRACKED

$$\frac{x \notin \{x_1, \ldots, x_n\}}{\mathcal{H}, \ell\langle x_1[F_1], \ldots, x_n[F_n] \rangle \vdash x@\ell \ \texttt{untracked}}$$

ASSIGNMENT-VALID

$$\frac{\mathcal{H} \vdash x@\ell \ \texttt{untracked}}{\mathcal{H} \vdash x@\ell = \ell'}$$

B.3.3 *Virtual Commands*

FOCUS-HEAP

$$\Gamma, x : \ell\ \tau \vdash \texttt{focus}\ \ell@x : \mathcal{H}, \ell\langle\rangle \Rightarrow \mathcal{H}, \ell\langle x[\,]\rangle$$

UNFOCUS-HEAP

$$\Gamma \vdash \texttt{unfocus}\ \ell@x : \mathcal{H}, \ell\langle x[], X\rangle \Rightarrow \mathcal{H}, \ell\langle X\rangle$$

EXPLORE-HEAP

$$\frac{F = f_1 \mapsto \ell_1, \ldots, f_n \mapsto \ell_n \qquad f \notin \{f_1, \ldots, f_n\} \qquad \ell' \textit{ fresh}}{\Gamma \vdash \texttt{explore}\ \ell@x.f : \mathcal{H}, \ell\langle x[F], X\rangle \Rightarrow \mathcal{H}', \ell\langle x[F, f \mapsto \ell'], X\rangle, \ell'\langle\,\rangle}$$

RETRACT-HEAP

$$\Gamma \vdash \texttt{retract}\ \ell' : \mathcal{H}, \ell\langle x[F, f \mapsto \ell'], X\rangle, \ell'\langle\rangle \Rightarrow \mathcal{H}, \ell\langle x[F], X\rangle$$

ATTACH-HEAP

$$\frac{\mathcal{H}' = \mathcal{H}[\ell_1 \mapsto \ell_2] \qquad X_1' = X_1[\ell_1 \mapsto \ell_2] \qquad X_2' = X_2[\ell_1 \mapsto \ell_2]}{\vdash \texttt{attach}\ \ell_1\ \texttt{to}\ \ell_2 : \mathcal{H}, \ell_1\langle X_1\rangle, \ell_2\langle X_2\rangle \Rightarrow \mathcal{H}', \ell_2\langle X_1', X_2'\rangle}$$

ATTACH-HEAP-NOOP

$$\Gamma \vdash \texttt{attach}\ \ell\ \texttt{to}\ \ell : \mathcal{H} \Rightarrow \mathcal{H}$$

B.3.4 *Functions*

CONSUMES

$$\Gamma \vdash \texttt{consumes}\ \ell : \mathcal{H}, \ell\langle\rangle \Rightarrow \mathcal{H}$$

PRESERVES

$$\Gamma \vdash \texttt{preserves}\ \ell : \mathcal{H}, \ell\langle\rangle \Rightarrow \mathcal{H}, \ell\langle\rangle$$

PRESERVE-ISOLATED-FUNC

$$\frac{\Gamma \vdash q\ \ell : \mathcal{H} \Rightarrow \mathcal{H}' \qquad \ell' \notin \mathit{region\text{-}names}(\Gamma, \mathcal{H}')}{\Gamma \vdash (q\ \ell \to \texttt{isolated}\ \ell') : \mathcal{H} \Rightarrow \mathcal{H}', \ell'\langle\,\rangle}$$

CONSUME-FUNC

$$\frac{\Gamma \vdash (\texttt{consumes}\ \ell \to \texttt{isolated}\ \ell') : \mathcal{H} \Rightarrow \mathcal{H}'}{\Gamma \vdash (\texttt{consumes}\ \ell \to \cdot\ \ell') : \mathcal{H} \Rightarrow \mathcal{H}'}$$

PRESERVE-SIMPLE-FUNC

$$\frac{\Gamma \vdash \texttt{preserves}\ \ell : \mathcal{H} \Rightarrow \mathcal{H}}{\Gamma \vdash (\texttt{preserves}\ \ell \to \cdot\ \ell) : \mathcal{H} \Rightarrow \mathcal{H}}$$

B.4 META-RULES

B.4.1 *Equivalences*

REGION-α-EQUIVALENCE

$$\frac{m = (\ell_1 \mapsto \ell_1', \ldots, \ell_n \mapsto \ell_n') \text{ is a bijection from } \mathit{regions}(\mathcal{H}) \text{ to } \mathit{regions}(\mathcal{H}') \qquad \mathcal{H} = \ell\langle x[f \mapsto \ell'', \ldots], \ldots\rangle, \ldots}{\mathcal{H}, \Gamma \equiv_m m(\ell)\langle x[f \mapsto m(\ell''), \ldots], \ldots\rangle, \ldots; \{x : m(\ell)\tau \mid x : \ell\tau \in \Gamma\}}$$

B.5 DYNAMIC SEMANTICS

B.5.1 *Evaluation Contexts*

$$
\begin{aligned}
R_d &: 2^l \\
\pi &: l \rightharpoonup o \\
o &: (\tau, v) \\
\sigma &: x \rightharpoonup l
\end{aligned}
$$

$$
\begin{aligned}
E ::= [\cdot] \;&|\; E \oplus e \mid l \oplus E \mid E; e \mid l; E \mid \texttt{if}\ (E)\ \{e\}\ \texttt{else}\ \{e\} \\
&|\; \texttt{while}\ (E)\ \{e\} \mid x = E \mid E(e) \mid l(E) \mid E.f \\
&|\; \texttt{send}\ E \mid \texttt{receive}\ E \mid \texttt{attach}\ E\ e \mid \texttt{attach}\ l\ E \\
&|\; \texttt{focus}\ E \mid \texttt{unfocus}\ E \mid \texttt{explore}\ E \mid \texttt{retract}\ E
\end{aligned}
$$

$$
\frac{(R_d, \pi, \sigma, e) \xrightarrow{eval} (R'_d, \pi', \sigma', e')}{(R_d, \pi, \sigma, E[e]) \xrightarrow{eval} (R'_d, \pi', \sigma', E[e'])}
$$

B.5.2 *Small-Step Semantics*

$$
\pi \vdash l \hookrightarrow l \qquad \frac{\pi(l) = (\tau, v) \qquad v[f] = l'}{\pi \vdash l \hookrightarrow l'} \qquad \frac{\pi \vdash l \hookrightarrow l' \qquad \pi \vdash l' \hookrightarrow l''}{\pi \vdash l \hookrightarrow l''}
$$

$$\frac{\sigma(x) \in R_d}{(R_d, \pi, \sigma, x) \xrightarrow{eval} (R_d, \pi, \sigma, \sigma(x))}$$

$$\frac{l \textit{ fresh}}{(R_d, \pi, \sigma, constant) \xrightarrow{eval} (R_d \cup \{l\}, \pi[l \mapsto (typeof(constant), constant)], \sigma, l)}$$

$$\frac{l_l, l_r \in R_d \qquad \pi(l_l) = o_l}{(R_d, \pi, \sigma, l_l; l_r) \xrightarrow{eval} (R_d, \pi, \sigma, l_r)}$$

$$\frac{l_l, l_r \in R_d \qquad \pi(l_l) = o_l \qquad \pi(l_r) = o_r}{(R_d, \pi, \sigma, l_l \oplus l_r) \xrightarrow{eval} (R_d, \pi, \sigma, [\![\oplus]\!](o_l, o_r))}$$

$$\frac{l \in R_d \qquad \pi(l) = \texttt{true}}{(R_d, \pi, \sigma, \texttt{if}(l)\{e\}\ \texttt{else}\ \{e_i\}) \xrightarrow{eval} (R_d, \pi, \sigma, e)}$$

$$\frac{l \in R_d \qquad \pi(l) = \texttt{false}}{(R_d, \pi, \sigma, \texttt{if}(l)\{e_i\}\ \texttt{else}\ \{e\}) \xrightarrow{eval} (R_d, \pi, \sigma, e)}$$

$$(R_d, \pi, \sigma, \texttt{while}\ (e_1)\ \{e_2\}) \xrightarrow{eval} (R_d, \pi, \sigma, \texttt{if}(e_1)\ \{e_2; \texttt{while}\ (e_1)\ \{e_2\}\})$$

$$(R_d, \pi, \sigma, \texttt{declare}\ x : \tau\ \texttt{in}\ e) \xrightarrow{eval} (R_d, \pi, \sigma, e)$$

$$(R_d, \pi, \sigma, x = l) \xrightarrow{eval} (R_d, \pi, \sigma[x \mapsto l], l)$$

$$\frac{\begin{array}{c} l_f \in R_d \qquad \pi(l_f) = (\tau_f, v_f) \\ F_d(v_f) = \lambda x.e \qquad e \equiv_\alpha e' \qquad FV(e') = \{x'\} \qquad x' \textit{ fresh} \end{array}}{(R_d, \pi, \sigma, l_f(l)) \xrightarrow{eval} (R_d, \pi, \sigma, \{x' = l; e'\})}$$

$$\frac{dom(\pi') = R'_d \qquad \vdash \pi' \qquad \pi'(l) = (\tau, v) \qquad \forall l' \in dom(\pi').\ \ \pi' \vdash l \hookrightarrow l'}{(R_d \uplus R'_d, \pi \uplus \pi', \sigma, \texttt{send}\ \tau(l)) \xrightarrow{eval} (R_d, \pi \uplus \pi', \sigma, l)}$$

$$\frac{\begin{array}{c} dom(\pi') \cap R_d = \emptyset \\ \vdash \pi' \qquad \pi'(l) = (\tau, v) \qquad \forall l' \in dom(\pi').\ \ \pi' \vdash l \hookrightarrow l' \end{array}}{(R_d, \pi \uplus \pi', \sigma, \texttt{receive}\ \tau()) \xrightarrow{eval} (R_d \cup dom(\pi'), \pi \uplus \pi', \sigma, l)}$$

$$\frac{l \in R_d \qquad \pi(l) = (\tau, v) \qquad v[f] = l_2}{(R_d, \pi, \sigma, l.f) \xrightarrow{eval} (R_d, \pi, \sigma, l_2)}$$

$$\frac{l \in R_d \qquad \pi(l) = (\tau, v) \qquad q_r\ f \in \mathit{fields}(\tau)}{(R_d, \pi, \sigma, l.f = l_2) \xrightarrow{eval} (R_d, \pi[l \mapsto (\tau, v[f \mapsto l_2])], \sigma, l_2)}$$

$$\frac{cmd \in \{\texttt{attach}\ l\ l_i, \texttt{focus}\ l, \texttt{unfocus}\ l, \texttt{explore}\ l, \texttt{retract}\ l\}}{(R_d, \pi, \sigma, cmd) \xrightarrow{eval} (R_d, \pi, \sigma, l)}$$

$$[\![+]\!]((\texttt{int}, n_1), (\texttt{int}, n_2)) \triangleq n_1 + n_2$$

$$[\![*]\!]((\texttt{int}, n_1), (\texttt{int}, n_2)) \triangleq n_1 * n_2$$

$$[\![-]\!]((\texttt{int}, n_1), (\texttt{int}, n_2)) \triangleq n_1 - n_2$$

B.6 CONFIGURATION TYPING

B.6.1 *Locations in the Heap*

$$\frac{\begin{array}{c}\forall l \in dom(\pi), \tau, v \text{ s.t. } l = (\tau, v)\text{: } \forall f, l' \text{ s.t. } v[f] = l'\text{:} \\ \exists q_r, \tau_f, v_f \text{ s.t. } q_r\ f\ \tau_f \in \mathit{fields}(\tau) \wedge \pi(l') = (\tau_f, v_f)\end{array}}{\vdash \pi}$$

$$\frac{\pi(l) = (\tau, v) \qquad q_r\ f \in \mathit{fields}(\tau) \qquad v[f] = l'}{\pi \vdash l[f] = l'}$$

$$\frac{\begin{array}{c}\forall l : (l \in dom(\pi)) \Rightarrow \exists \ell, \tau, v\text{: } (\mathrm{P} \vdash l : \ell\ \tau) \wedge (\pi(l) = (\tau, v)) \\ \forall l, l', f\text{: } (\pi \vdash l[f] = l') \Rightarrow \exists \ell, \tau, \tau'\text{: } \mathrm{P} \vdash l : \ell\ \tau \wedge ((\mathrm{P} \vdash l' : \ell\ \tau') \vee (\texttt{isolated}\ f : \tau' \in \mathit{fields}(\tau)))\end{array}}{\vdash \pi : \mathrm{P}}$$

B.6.2 *Simplicity*

$$\frac{\forall l, \tau \; s.t. \; \mathrm{P} \vdash l : \ell \; \tau\text{:} \;\; \pi; \mathrm{P} \vdash l \; \texttt{simple} \qquad \vdash \pi : \mathrm{P}}{\pi; \mathrm{P} \vdash \ell \; \texttt{simple}}$$

$$\frac{\begin{array}{c} \pi(l) = (\tau, v) \qquad \texttt{isolated}\; f \in \tau \\ \mathrm{P} \vdash l : \ell \; \tau \qquad \mathrm{P} \vdash v[f] : \ell' \; \tau' \qquad \ell \neq \ell' \qquad \pi; \mathrm{P} \vdash \ell' \; \texttt{dominated} \\ \pi; \mathrm{P} \vdash \ell' \; \texttt{simple} \qquad \pi; \mathrm{P} \vdash \ell' \; \texttt{no-cycles} \qquad \vdash \pi : \mathrm{P} \end{array}}{\pi; \mathrm{P} \vdash l.f \; \texttt{simple}}$$

$$\frac{\pi(l) = (\tau, v) \qquad \cdot f \in \tau}{\pi; \mathrm{P} \vdash l.f \; \texttt{simple}}$$

$$\frac{\pi(f) = (\tau, v) \qquad \forall q_r \; f \in \mathit{fields}(\tau)\text{:} \;\; \pi; \mathrm{P} \vdash l.f \; \texttt{simple}}{\pi; \mathrm{P} \vdash l \; \texttt{simple}}$$

π;P $\vdash \ell$ `dominated` if:

$\forall l_1, l_2, l_3, l_4$ where :

$$\begin{array}{rr} \mathrm{P} \vdash l_1 : \ell_1 \tau_1 & \text{for some } \tau_1 \text{ and } \ell_1 \neq \ell \\ \wedge \mathrm{P} \vdash l_2 : \ell \tau_2 & \text{for some } \tau_2 \\ \wedge \mathrm{P} \vdash l_3 : \ell_3 \tau_3 & \text{for some } \tau_3 \text{ and } \ell_3 \neq \ell \\ \wedge \mathrm{P} \vdash l_4 : \ell \tau_4 & \text{for some } \tau_4 \\ \wedge \pi \vdash l_1[f_1] = l_2 & \text{for some } f_1 \\ \wedge \pi \vdash l_3[f_3] = l_4 & \text{for some } f_3 \end{array}$$

Then $l_1 = l_3 \wedge f_1 = f_3$.

$$\frac{\pi(l) = (\tau, v) \qquad \mathrm{P} \vdash l : \ell\ \tau}{\mathrm{P}; \pi \vdash l : \ell\ \tau} \qquad \frac{\mathrm{P}; \pi \vdash l : \ell\ \tau \qquad \mathrm{P}; \pi \vdash l' : \ell\ \tau'}{\mathrm{P}; \pi \vdash l\ \texttt{->}l'}$$

$$\frac{\begin{array}{c}\mathrm{P}; \pi \vdash l : \ell\ \tau \qquad \mathrm{P}; \pi \vdash l' : \ell'\ \tau' \\ \pi(l) = (\tau, v) \qquad v[f] = l' \qquad \texttt{isolated}\ f : \tau' \in \mathit{fields}(\tau)\end{array}}{\mathrm{P}; \pi \vdash l\ \texttt{->}l'}$$

$$\frac{\mathrm{P}; \pi \vdash l \texttt{->} l' \qquad \mathrm{P}; \pi \vdash l' \texttt{->} l''}{\mathrm{P}; \pi \vdash l \texttt{->} l''}$$

$\pi; \mathrm{P} \vdash \ell$ `no-cycles` if $\forall l, l', \ell, \ell', \tau, \tau'$ where

$$\mathrm{P} \vdash l : \ell\ \tau \wedge \mathrm{P} \vdash l' : \ell'\ \tau' \wedge \mathrm{P}; \pi \vdash l\ \texttt{->}\ l' :$$

$$(\mathrm{P}; \pi \vdash l'\ \texttt{->}\ l) \Rightarrow (\ell = \ell')$$

$$\frac{\begin{array}{c}\sigma(x) = l \qquad \mathrm{P} \vdash l : \ell\ \tau \\ \pi(l) = (\tau, v) \qquad \forall f \mapsto \ell' \in F\colon (f \in \mathit{fields}(\tau)) \wedge \mathrm{P} \vdash v[f] : \ell' \\ \forall f \in (\mathit{fields}(\tau) - \mathit{fields}(F))\colon \pi; \mathrm{P} \vdash v[f]\ \texttt{simple} \\ \forall l' \neq l\colon (\mathrm{P} \vdash l' : \ell\ \tau') \Rightarrow (\pi; \mathrm{P} \vdash l'\ \texttt{simple}) \qquad \sigma, \mathrm{P} \vdash \pi : \mathcal{H} \qquad \vdash \pi : \mathrm{P}\end{array}}{\sigma, \mathrm{P} \vdash \pi : \ell\langle x[F]\rangle, \mathcal{H}}$$

$$\frac{\forall l\colon (\mathrm{P} \vdash l : \ell\ \tau) \Rightarrow (\pi; \mathrm{P} \vdash l\ \texttt{simple}) \qquad \sigma, \mathrm{P} \vdash \pi : \mathcal{H} \qquad \vdash \pi : \mathrm{P}}{\sigma, \mathrm{P} \vdash \pi : \ell\langle\rangle, \mathcal{H}}$$

B.6.3 *Stack Typing*

$$\frac{\vdash \pi : \mathrm{P}}{\mathrm{P} \vdash (\pi, \sigma) : \cdot} \qquad \frac{\vdash \pi : \mathrm{P} \qquad \vdash (\pi, \sigma) : \Gamma \qquad \mathrm{P} \vdash \sigma(x) : \ell\ \tau}{\mathrm{P} \vdash (\pi, \sigma) : (x : \ell\ \tau, \Gamma)}$$

B.6.4 *Configuration Typing*

$$\frac{\vdash \pi \qquad \vdash \mathcal{H}; \Gamma \qquad \vdash \pi : \mathrm{P} \qquad \mathrm{P} \vdash (\pi, \sigma) : \Gamma \qquad \sigma, \mathrm{P} \vdash \pi : \mathcal{H}}{\vdash \pi, \sigma : \mathcal{H}; \Gamma; \mathrm{P}}$$

$$\frac{\forall \ell \text{ s.t. } \mathcal{H} \vdash \ell\ .\ \forall l, \tau \text{ s.t. } \mathrm{P} \vdash l : \ell\ \tau\ .\ l \in R_d \qquad \vdash \pi : \mathrm{P} \qquad R_d \subseteq dom(\pi)}{\pi; \mathrm{P} \vdash R_d : \mathcal{H}}$$

$$\frac{\vdash \pi, \sigma : \mathcal{H}; \Gamma; \mathrm{P} \qquad \pi; \mathrm{P} \vdash R_d : \mathcal{H}}{\vdash (R_d, \pi, \sigma) : (\mathcal{H}; \Gamma; \mathrm{P})}$$

$$\frac{\vdash (R_d, \pi, \sigma) : (\mathcal{H}; \Gamma; \mathrm{P}) \qquad \mathcal{H}; \Gamma; \mathrm{P} \vdash e \dashv \Gamma'; \mathcal{H}'}{\vdash (R_d, \pi, \sigma, e)}$$

B.7 PROGRESS AND PRESERVATION

Theorem B.1 (Progress). *For any (R_d, π, σ, e) where e is not a value, if $\vdash (R_d, \pi, \sigma, e)$ then there exists some $(R'_d, \pi', \sigma', e')$ such that $(R_d, \pi, \sigma, e) \xrightarrow{eval} (R'_d, \pi', \sigma', e')$*

Theorem B.2 (Preservation). *For any* (R_d, π, σ, e)*, if* $\vdash (R_d, \pi, \sigma, e)$ *and there exists some* $(R'_d, \pi', \sigma', e')$ *where* $(R_d, \pi, \sigma, e) \xrightarrow{eval} (R'_d, \pi', \sigma', e')$ *then* $\vdash (R'_d, \pi', \sigma', e')$

BIBLIOGRAPHY

Abelson, Harold and Gerald Jay Sussman (July 1996). *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press. ISBN: 0262011530. URL: https://www.xarg.org/ref/a/0262011530/ (cit. on p. 149).

Adya, A. (Mar. 1999). "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions." PhD thesis. Cambridge, MA: Massachusetts Institute of Technology (cit. on p. 68).

Agha, Gul (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-01092-5 (cit. on p. 94).

Aguilera, Marcos K., Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi (2019). "The Impact of RDMA on Agreement." In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. PODC '19. Toronto ON, Canada: Association for Computing Machinery, pp. 409–418. ISBN: 9781450362177. DOI: 10.1145/3293611.3331601. URL: https://doi.org/10.1145/3293611.3331601 (cit. on p. 13).

Aiken, Alex, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi (2003). "Checking and Inferring Local Non-Aliasing." In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. San Diego, California, USA: Association for Computing Machinery, pp. 129–140. ISBN: 1581136625. DOI: 10.1145/

781131.781146. URL: https://doi.org/10.1145/781131.781146 (cit. on p. 267).

Akkoorath, D. D., A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro (2016). "Cure: Strong Semantics Meets High Availability and Low Latency." In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 405–414 (cit. on pp. 20, 147, 160, 161).

Aldrich, Jonathan, Valentin Kostadinov, and Craig Chambers (2002). "Alias annotations for program understanding." In: *17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. Seattle, Washington, USA, pp. 311–330. ISBN: 1-58113-471-1 (cit. on pp. 27, 188, 258, 259).

Almeida, Paulo Sérgio (1997). "Balloon types: Controlling sharing of state in data types." In: *ECOOP'97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 32–59. ISBN: 978-3-540-69127-3 (cit. on pp. 188, 258).

Alvaro, Peter, Peter Bailis, Neil Conway, and Joseph M. Hellerstein (2013). "Consistency without borders." en. In: *ACM Symp. on Cloud Computing (SoCC)*, 23:1–23:10. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523632. URL: http://dl.acm.org/citation.cfm?doid=2523616.2523632 (cit. on pp. 105, 142, 179).

Alvaro, Peter, Neil Conway, Joseph M Hellerstein, and William R Marczak (2011). "Consistency Analysis in Bloom: a CALM and Collected Approach." In: *CIDR (Conference on Innovative Data Systems Research)*, pp. 249–260 (cit. on p. 142).

Ardekani, Masoud Saeida, Pierre Sutra, and Marc Shapiro (2013). "Non-Monotonic Snapshot Isolation: Scalable and Strong Consistency for

Geo-Replicated Transactional Systems." In: *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*. IEEE, pp. 163–172 (cit. on p. 88).

Askarov, Aslan and Andrew Myers (2010). "A Semantic Framework for Declassification and Endorsement." In: *Programming Languages and Systems*. Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 64–84. ISBN: 978-3-642-11957-6 (cit. on p. 8).

Atkinson, Malcolm and Ronald Morrison (July 1995). "Orthogonally Persistent Object Systems." In: *The VLDB Journal* 4.3, pp. 319–402. ISSN: 1066-8888. URL: http://dl.acm.org/citation.cfm?id=615224.615226 (cit. on p. 146).

Badros, Greg J (2000). "JavaML: a markup language for Java source code." In: *Computer Networks* 33.1, pp. 159–177. ISSN: 1389-1286. DOI: https://doi.org/10.1016/S1389-1286(00)00037-2. URL: http://www.sciencedirect.com/science/article/pii/S1389128600000372 (cit. on p. 167).

Bailis, Peter, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica (2014). "Coordination avoidance in database systems." In: *Proceedings of the VLDB Endowment* 8.3, pp. 185–196 (cit. on p. 180).

Bailis, Peter, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica (2013). "Bolt-on causal consistency." In: *ACM SIGMOD Int'l Conf. on Management of Data* (cit. on pp. 76, 90).

Balakrishnan, Mahesh, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis (2012). "CORFU: A Shared Log Design for Flash Clusters." In: *9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. San Jose, CA, pp. 1–14. ISBN: 978-931971-

92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan (cit. on pp. 13, 14, 21, 140).

Balegas, Valter, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro (2015). "Putting consistency back into eventual consistency." In: *ACM SIGOPS/EuroSys European Conference on Computer Systems*, p. 6 (cit. on pp. 149, 168, 180).

Ban, Bela (Nov. 2002). *JGroups reliable multicast library*. http://jgroups.org/ (cit. on p. 141).

Banerjee, Anindya and David A. Naumann (2002). "Representation Independence, Confinement and Access Control [Extended Abstract]." In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. Portland, Oregon: Association for Computing Machinery, pp. 166–177. ISBN: 1581134509. DOI: 10.1145/503272.503289. URL: https://doi.org/10.1145/503272.503289 (cit. on p. 259).

Batty, Mark, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber (2011). "Mathematizing C++ Concurrency." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 38th ACM Symp. on Principles of Programming Languages (POPL). Austin, Texas, USA: Association for Computing Machinery, pp. 55–66. ISBN: 9781450304900. DOI: 10.1145/1926385.1926394. URL: https://doi.org/10.1145/1926385.1926394 (cit. on p. 3).

Beckman, Nels E., Kevin Bierhoff, and Jonathan Aldrich (2008). "Verifying Correct Usage of Atomic Blocks and Typestate." In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: Association for Computing Machinery, pp. 227–244. ISBN: 9781605582153. DOI:

10.1145/1449764.1449783. URL: https://doi.org/10.1145/1449764.1449783 (cit. on pp. 261, 263).

Behrens, Jonathan, Ken Birman, Sagar Jha, and Edward Tremel (2018). "RDMC: A Reliable RDMA Multicast for Large Objects." In: *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DSN '18. Washington, DC, USA: IEEE Computer Society, pp. 1–12 (cit. on pp. 17, 123, 125).

Belay, Adam, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion (Oct. 2014). "IX: A Protected Dataplane Operating System for High Throughput and Low Latency." In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, pp. 49–65. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay (cit. on p. 105).

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil (May 1995). "A Critique of ANSI SQL Isolation Levels." In: *ACM SIGMOD Int'l Conf. on Management of Data* (cit. on pp. 49, 69).

Biba, K. J. (Apr. 1977). *Integrity Considerations for Secure Computer Systems*. Tech. rep. ESD-TR-76-372. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.) Bedford, MA: USAF Electronic Systems Division (cit. on pp. 8, 35, 51).

Birman, K. and T. Joseph (1987). "Exploiting Virtual Synchrony in Distributed Systems." In: *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. SOSP '87. Austin, Texas, USA: ACM, pp. 123–138. ISBN: 0-89791-242-X. DOI: 10.1145/41457.37515. URL: http://doi.acm.org/10.1145/41457.37515 (cit. on p. 17).

Birman, Kenneth P. (1985). "Replication and Fault-tolerance in the ISIS System." In: *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*. SOSP '85. Orcas Island, Washington, USA: ACM, pp. 79–86. ISBN: 0-89791-174-1. DOI: 10.1145/323647.323636. URL: http://doi.acm.org/10.1145/323647.323636 (cit. on p. 141).

Birman, Kenneth P. (2012). *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. New York, NY, USA: Springer Verlag Texts in Computer Science. ISBN: 1447124154 (cit. on pp. 118, 139).

Birman, Kenneth P. and Thomas A. Joseph (Jan. 1987). "Reliable Communication in the Presence of Failures." In: *ACM Trans. Comput. Syst.* 5.1, pp. 47–76. ISSN: 0734-2071. DOI: 10.1145/7351.7478. URL: http://doi.acm.org/10.1145/7351.7478 (cit. on pp. 94, 118, 121).

Boyapati, Chandrasekhar, Robert Lee, and Martin Rinard (Nov. 2002). "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks." In: *SIGPLAN Not.* 37.11, pp. 211–230. ISSN: 0362-1340. DOI: 10.1145/583854.582440. URL: https://doi.org/10.1145/583854.582440 (cit. on p. 259).

Boyapati, Chandrasekhar and Martin Rinard (Oct. 2001). "A parameterized type system for race-free Java programs." In: *16th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. Tampa Bay, FL (cit. on pp. 26, 27, 258, 259, 263).

Boyland, John (2001). "Alias burying: Unique variables without destructive reads." In: *Software: Practice and Experience* 31.6, pp. 533–553. DOI: 10.1002/spe.370 (cit. on p. 259).

Boyland, John Tang (Aug. 2010). "Semantics of Fractional Permissions with Nesting." In: *ACM Trans. Program. Lang. Syst.* 32.6. ISSN: 0164-0925. DOI:

10.1145/1749608.1749611. URL: https://doi.org/10.1145/1749608.1749611 (cit. on p. 267).

Boyland, John Tang and William Retert (2005). "Connecting Effects and Uniqueness with Adoption." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: Association for Computing Machinery, pp. 283–295. ISBN: 158113830X. DOI: 10.1145/1040305.1040329. URL: https://doi.org/10.1145/1040305.1040329 (cit. on p. 263).

Boyland, John, James Noble, and William Retert (2001). "Capabilities for Sharing." In: *ECOOP 2001 — Object-Oriented Programming*. Ed. by Jørgen Lindskov Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 2–27. ISBN: 978-3-540-45337-6 (cit. on p. 259).

Brewer, Eric (2010). "A Certain Freedom: Thoughts on the CAP Theorem." In: *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. PODC '10. Zurich, Switzerland: ACM, pp. 335–335. ISBN: 978-1-60558-888-9. DOI: 10.1145/1835698.1835701. URL: http://doi.acm.org/10.1145/1835698.1835701 (cit. on pp. 2, 34, 102).

Brown, Jane W S (Nov. 15, 2017). Personal communication. Google, Inc. (cit. on pp. 6, 34).

Brutschy, Lucas, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev (Jan. 2017). "Serializability for Eventual Consistency: Criterion, Analysis, and Applications." In: *44th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 458–472 (cit. on pp. 34, 88, 89).

Burckhardt, Sebastian, Alexandro Baldassin, and Daan Leijen (2010). "Concurrent Programming with Revisions and Isolation Types." In: *25th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. OOPSLA '10. Reno/Tahoe, Nevada, USA,

pp. 691–707. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869515. URL: http://doi.acm.org/10.1145/1869459.1869515 (cit. on p. 180).

Burckhardt, Sebastian, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood (2012). "Cloud types for eventual consistency." In: *European Conference on Object-Oriented Programming*. Springer, pp. 283–307 (cit. on p. 179).

Calder, Brad et al. (2011). "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: ACM, pp. 143–157. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043571. URL: http://doi.acm.org/10.1145/2043556.2043571 (cit. on p. 141).

Carlson, Josiah L. (2013). *Redis in Action*. Greenwich, CT, USA: Manning Publications Co. ISBN: 978-1617290855 (cit. on pp. 6, 33).

Castegren, Elias and Tobias Wrigstad (2016). "Reference Capabilities for Concurrency Control." In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 5:1–5:26. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.5. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6099 (cit. on pp. 259, 263).

Chandra, Tushar D., Robert Griesemer, and Joshua Redstone (2007). "Paxos Made Live: An Engineering Perspective." In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC '07. New York, NY, USA: ACM, pp. 398–407. ISBN: 978-1-59593-

616-5. DOI: 10.1145/1281100.1281103. URL: http://doi.acm.org/10.1145/1281100.1281103 (cit. on p. 139).

Chandra, Tushar Deepak and Sam Toueg (Mar. 1996). "Unreliable Failure Detectors for Reliable Distributed Systems." In: *J. ACM* 43.2, pp. 225–267. ISSN: 0004-5411. DOI: 10.1145/226643.226647. URL: http://doi.acm.org/10.1145/226643.226647 (cit. on p. 122).

Chang, Jo-Mei and N. F. Maxemchuk (Aug. 1984). "Reliable Broadcast Protocols." In: *ACM Trans. Comput. Syst.* 2.3, pp. 251–273. ISSN: 0734-2071. DOI: 10.1145/989.357400. URL: http://doi.acm.org/10.1145/989.357400 (cit. on p. 140).

Chatterjee, Shankha and Wojciech Golab (2017). "Self-Tuning Eventually-Consistent Data Stores." In: *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, pp. 78–92 (cit. on p. 89).

Cheung, Alvin, Samuel Madden, Owen Arden, and Andrew C Myers (Aug. 2012). "Automatic Partitioning of Database Applications." In: *PVLDB* 5.11, pp. 1471–1482. URL: http://vldb.org/pvldb/vol5/p1471_alvincheung_vldb2012.pdf (cit. on p. 53).

Clancy, Kevin and Heather Miller (2017). "Monotonicity Types for Distributed Dataflow." In: *Proceedings of the 2nd Workshop on Programming Models and Languages for Distributed Computing*. CONF (cit. on p. 183).

Clarke, Dave and Tobias Wrigstad (2003). "External Uniqueness Is Unique Enough." In: *ECOOP 2003 – Object-Oriented Programming*. Ed. by Luca Cardelli. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 176–200. ISBN: 978-3-540-45070-2 (cit. on p. 258).

Clarke, Dave, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen (2008). "Minimal ownership for active objects." In: *Asian Symposium on*

Programming Languages and Systems. Springer, pp. 139–154 (cit. on pp. 26, 258, 259).

Clarke, David G, John M Potter, and James Noble (1998). "Ownership types for flexible alias protection." In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, pp. 48–64. ISBN: 1581130058. DOI: 10.1145/286936.286947. URL: https://doi.org/10.1145/286936.286947 (cit. on pp. 26, 257).

Clebsch, Sylvan, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil (2015). "Deny capabilities for safe, fast actors." en. In: *5th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)* Pp. 1–12. ISBN: 978-1-4503-3901-8. DOI: 10.1145/2824815.2824816. URL: http://dl.acm.org/citation.cfm?doid=2824815.2824816 (cit. on pp. 27, 259, 263, 269).

Cok, David R. (2011). "OpenJML: JML for Java 7 by Extending OpenJDK." In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 472–479. ISBN: 978-3-642-20398-5 (cit. on p. 168).

Conway, Neil, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier (2012). "Logic and Lattices for Distributed Programming." In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 1:1–1:14. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391230. URL: http://doi.acm.org/10.1145/2391229.2391230 (cit. on pp. 12, 142, 273).

Cooper, Brian F, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver,

and Ramana Yerneni (2008). "PNUTS: Yahoo!'s hosted data serving platform." In: *Proceedings of the VLDB Endowment* 1.2, pp. 1277–1288 (cit. on p. 89).

Corbett, James C, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. (2013). "Spanner: Google's globally distributed database." In: *ACM Transactions on Computer Systems (TOCS)* 31.3, p. 8 (cit. on pp. 14, 37).

Gentz, Mimi, Aravind Krishna R, Luis Bosquez, Mark McGee, Tyson Nevil, Kris Crider, Yaron Y. Goland, Andy Pasic, and Ji Huang Carol Zeumault (2017). *Welcome to Azure Cosmos DB*. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. URL: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction (cit. on pp. 6, 33, 89).

Crary, Karl, David Walker, and Greg Morrisett (1999). "Typed Memory Management in a Calculus of Capabilities." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, pp. 262–275. ISBN: 1581130953. DOI: 10.1145/292540.292564. URL: https://doi.org/10.1145/292540.292564 (cit. on p. 262).

Crooks, Natacha, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement (2016). "Tardis: A branch-and-merge approach to weak consistency." In: *ACM SIGMOD Int'l Conf. on Management of Data*, pp. 1615–1628 (cit. on pp. 33, 38, 76, 179).

Cui, Heming, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang (2015). "Paxos Made Transparent." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, pp. 105–

120. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815427. URL: http://doi.acm.org/10.1145/2815400.2815427 (cit. on p. 139).

Cunningham, David, Sophia Drossopoulou, and Susan Eisenbach (2007). "Universes for race safety." In: *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pp. 20–51 (cit. on p. 259).

Dang, Huynh Tu, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé (2015). "NetPaxos: Consensus at Network Speed." In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. Santa Clara, California: ACM, 5:1–5:7. ISBN: 978-1-4503-3451-8. DOI: 10.1145/2774993.2774999. URL: http://doi.acm.org/10.1145/2774993.2774999 (cit. on pp. 13, 141).

DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels (2007). "Dynamo: Amazon's Highly Available Key–Value Store." In: *21st ACM Symp. on Operating System Principles (SOSP)* (cit. on pp. 20, 89, 179).

Degen, Markus, Peter Thiemann, and Stefan Wehr (2007). "Tracking Linear and Affine Resources with Java(X)." In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 550–574. ISBN: 978-3-540-73589-2 (cit. on p. 261).

DeLine, Robert and Manuel Fähndrich (2004). "Typestates for Objects." In: *ECOOP 2004 – Object-Oriented Programming*. Ed. by Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 465–490. ISBN: 978-3-540-24851-4 (cit. on pp. 261, 263).

Dongol, Brijesh, Radha Jagadeesan, and James Riely (Jan. 2018). "Transactions in Relaxed Memory Architectures." In: *45th ACM Symp. on Principles of Programming Languages (POPL)*, 18:1–18:29 (cit. on pp. 34, 88, 89).

Dragojević, Aleksandar, Dushyanth Narayanan, Miguel Castro, and Orion Hodson (2014). "FaRM: Fast Remote Memory." In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, pp. 401–414. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%7B%5C'c%7D (cit. on pp. 13, 14, 142).

Du, Jiaqing, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel (2014). "Gentlerain: Cheap and Scalable Causal Consistency with Physical Clocks." In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM, pp. 1–13 (cit. on p. 88).

Dubey, Ayush, Greg D. Hill, Robert Escriva, and Emin Gün Sirer (2016). "Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps." In: *Proceedings of the VLDB Endowment* 9.11, pp. 852–863 (cit. on p. 77).

Duggan, Jennie, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik (2015). "The BigDAWG Polystore System." In: *ACM SIGMOD Record* 44.2, pp. 11–16 (cit. on p. 77).

Fähndrich, Manuel, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi (Apr. 2006). "Language Support for Fast and Reliable Message-Based Communication in Singularity OS." In: *SIGOPS Oper. Syst. Rev.* 40.4, pp. 177–190. ISSN: 0163-5980. DOI: 10.1145/1218063.1217953. URL: https://doi.org/10.1145/1218063.1217953 (cit. on p. 269).

Fähndrich, Manuel and Robert DeLine (June 2002). "Adoption and Focus: Practical Linear Types for Imperative Programming." In: *ACM SIGPLAN*

Conf. on Programming Language Design and Implementation (PLDI) (cit. on pp. 27, 189, 259, 261, 263, 265, 269).

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi (Feb. 2001). *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press. ISBN: 0262062186. URL: https://www.xarg.org/ref/a/0262062186/ (cit. on p. 149).

Fluet, Matthew and Greg Morrisett (2006). "Monadic Regions." In: *Journal of Functional Programming* 16.4-5, pp. 485–545. DOI: 10.1017/S095679680600596X (cit. on p. 263).

Fluet, Matthew, Greg Morrisett, and Amal Ahmed (2006). "Linear regions are all you need." In: *European Symposium on Programming*. Springer, pp. 7–21 (cit. on pp. 26, 27, 263).

Foster, Jeffrey S., Tachio Terauchi, and Alex Aiken (2002). "Flow-Sensitive Type Qualifiers." In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI '02. Berlin, Germany: Association for Computing Machinery, pp. 1–12. ISBN: 1581134630. DOI: 10.1145/512529.512531. URL: https://doi.org/10.1145/512529.512531 (cit. on p. 267).

Friedman, Roy and Robbert van Renesse (1997). "Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols." In: *Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC '97)*. Also available as *Technical Report 95-1527, Department of Computer Science, Cornell University.* (Cit. on p. 141).

Gallaire, Hervé and Jack Minker, eds. (1978). *Logic and data bases*. New York: Plenum Press (cit. on p. 94).

Gao, Lei, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar (2003). "Application-Specific Data Replication for Edge Services." In:

Proceedings of the 12th international conference on World Wide Web. ACM, pp. 449–460 (cit. on p. 89).

Gilbert, Seth and Nancy Lynch (June 2002). "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services." In: *SIGACT News* 33.2, pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601 (cit. on p. 102).

Girard, Jean-Yves (1987). "Linear Logic." In: *Theoretical Computer Science* 50.1, pp. 1–101. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90045-4. URL: http://dx.doi.org/10.1016/0304-3975(87)90045-4 (cit. on p. 261).

Goguen, Joseph A. and Jose Meseguer (Apr. 1982). "Security Policies and Security Models." In: *IEEE Symp. on Security and Privacy*, pp. 11–20 (cit. on p. 50).

Gordon, Colin S, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy (2012). "Uniqueness and reference immutability for safe parallelism." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. Tucson, Arizona, USA: Association for Computing Machinery, pp. 21–40. ISBN: 97814503156l6. DOI: 10.1145/2384616.2384619. URL: https://doi.org/10.1145/2384616.2384619 (cit. on pp. 188, 269).

Gotsman, Alexey, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro (Jan. 2016). "'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems." In: *43rd ACM Symp. on Principles of Programming Languages (POPL)*, pp. 371–384 (cit. on p. 88).

Gray, Jim (1981). "The transaction concept: Virtues and limitations." In: *Int'l Conf. on Very Large Data Bases (VLDB)*. Vol. 81, pp. 144–154 (cit. on p. 7).

Grossman, Dan, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney (May 2002). "Region-Based Memory Management in Cyclone." In: *SIGPLAN Not.* 37.5, pp. 282–293. ISSN: 0362-1340. DOI: 10.1145/543552.512563. URL: https://doi.org/10.1145/543552.512563 (cit. on p. 263).

Guerraoui, Rachid, Ron R. Levy, Bastian Pochon, and Vivien Quéma (July 2010). "Throughput Optimal Total Order Broadcast for Cluster Environments." In: *ACM Trans. Comput. Syst.* 28.2, 5:1–5:32. ISSN: 0734-2071. DOI: 10.1145/1813654.1813656. URL: http://doi.acm.org/10.1145/1813654.1813656 (cit. on p. 140).

Guerraoui, Rachid, Matej Pavlovic, and Dragos-Adrian Seredinschi (2016). "Incremental consistency guarantees for replicated objects." In: *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pp. 169–184 (cit. on pp. 89, 179).

Guo, Chuanxiong, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn (2016). "RDMA over Commodity Ethernet at Scale." In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, pp. 202–215. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934908. URL: http://doi.acm.org/10.1145/2934872.2934908 (cit. on p. 131).

Haerder, Theo and Andreas Reuter (1983). "Principles of Transaction-Oriented Database Recovery." In: *ACM Computing Surveys* 15, pp. 287–317 (cit. on p. 7).

Haller, Philipp and Martin Odersky (2010). "Capabilities for uniqueness and borrowing." In: *European Conference on Object-Oriented Programming*. Springer, pp. 354–378 (cit. on pp. 27, 259, 263, 268).

Halpern, Joseph Y. and Yoram Moses (July 1990). "Knowledge and Common Knowledge in a Distributed Environment." In: *J. ACM* 37.3, pp. 549–587. ISSN: 0004-5411. DOI: 10.1145/79147.79161. URL: http://doi.acm.org/10.1145/79147.79161 (cit. on p. 112).

Hamilton, Graham, Rick Cattell, Maydene Fisher, et al. (1997). *JDBC Database Access with Java*. Vol. 7. Addison Wesley (cit. on p. 90).

Helland, Pat and Dave Campbell (2009). "Building on Quicksand." en. In: *CIDR (Conference on Innovative Data Systems Research)* (cit. on p. 180).

Henglein, Fritz, Henning Makholm, and Henning Niss (2001). "A Direct Approach to Control-Flow Sensitive Region-Based Memory Management." In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '01. Florence, Italy: Association for Computing Machinery, pp. 175–186. ISBN: 158113388X. DOI: 10.1145/773184.773203. URL: https://doi.org/10.1145/773184.773203 (cit. on p. 262).

Herlihy, Maurice (Jan. 1991). "Wait-Free Synchronization." In: *ACM Transactions on Programming Languages* 1.13, pp. 124–149 (cit. on p. 88).

Herlihy, Maurice and Jeannette Wing (1988). *Linearizability: A Correctness Condition for Concurrent Objects*. Technical Report CMU-CS-88-120. Carnegie Mellon University, Pittsburgh, Pa. (cit. on pp. 2, 6, 37, 47, 70).

Hewitt, Carl, Peter Bishop, and Richard Steiger (1973). "A Universal Modular ACTOR Formalism for Artificial Intelligence." In: International Joint Conference on Artificial Intelligence (IJCAI). Stanford, CA, USA (cit. on pp. 18, 94, 182).

Hicks, Michael, Greg Morrisett, Dan Grossman, and Trevor Jim (2003). *Safe and flexible memory management in Cyclone*. Tech. rep. (cit. on p. 263).

Hogg, John (1991). "Islands: Aliasing Protection in Object-Oriented Languages." In: *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '91. Phoenix, Arizona, USA: Association for Computing Machinery, pp. 271–285. ISBN: 0201554178. DOI: 10.1145/117954.117975. URL: https://doi.org/10.1145/117954.117975 (cit. on pp. 188, 258).

Holt, Brandon, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze (2016). "Disciplined Inconsistency with Consistency Types." In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, pp. 279–293 (cit. on pp. 87, 88).

Holt, Brandon, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze (Apr. 2015). "Claret: Using Data Types for Highly Concurrent Distributed Transactions." In: *1st Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*, 4:1–4:4 (cit. on pp. 33, 88).

Houshmand, Farzin and Mohsen Lesani (2019). "Hamsaz: replication coordination analysis and synthesis." In: *ACM on Programming Languages (PACM)* 3.POPL, p. 74 (cit. on p. 179).

Hsu, Ta-Yuan and Ajay D. Kshemkalyani (2018). "Value the Recent Past: Approximate Causal Consistency for Partially Replicated Systems." In: *IEEE Transactions on Parallel and Distributed Systems* 29.1, pp. 212–225 (cit. on pp. 88, 90).

Jespersen, Thomas Bracht Laumann, Philip Munksgaard, and Ken Friis Larsen (2015). "Session Types for Rust." In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. WGP 2015. Vancouver, BC, Canada: Association for Computing Machinery, pp. 13–22. ISBN:

9781450338103. DOI: 10.1145/2808098.2808100. URL: https://doi.org/10.1145/2808098.2808100 (cit. on p. 264).

Jha, Sagar, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman (Apr. 2019). "Derecho: Fast State Machine Replication for Cloud Services." In: *ACM Trans. Comput. Syst.* 36.2. ISSN: 0734-2071. DOI: 10.1145/3302258. URL: https://doi.org/10.1145/3302258 (cit. on pp. 96, 99, 100, 103, 119, 125).

Jim, Trevor, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang (2002). "Cyclone: A Safe Dialect of C." In: *USENIX Annual Technical Conference, General Track*, pp. 275–288 (cit. on pp. 259, 262, 263).

Jung, Ralf, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer (Dec. 2017). "RustBelt: Securing the Foundations of the Rust Programming Language." In: *Proc. ACM Program. Lang.* 2.POPL, 66:1–66:34. ISSN: 2475-1421. DOI: 10.1145/3158154. URL: http://doi.acm.org/10.1145/3158154 (cit. on p. 264).

Kaki, Gowtham, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan (Dec. 2017). "Alone Together: Compositional Reasoning and Inference for Weak Isolation." In: *Proc. ACM Program. Lang.* 2.POPL. DOI: 10.1145/3158115. URL: https://doi.org/10.1145/3158115 (cit. on pp. 34, 88, 89).

Kalia, Anuj, Michael Kaminsky, and David G. Andersen (2014). "Using RDMA Efficiently for Key-value Services." In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, pp. 295–306. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.

2626299. URL: http://doi.acm.org/10.1145/2619239.2626299 (cit. on p. 143).

Kallman, Robert et al. (Aug. 2008). "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System." In: *PVLDB* 1.2 (cit. on p. 77).

Klophaus, Rusty (2010). "Riak Core: Building Distributed Applications Without Shared State." In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 14:1–14:1. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900176. URL: http://doi.acm.org/10.1145/1900160.1900176 (cit. on pp. 6, 33).

Kraska, Tim, Martin Hentschel, Gustavo Alonso, and Donald Kossmann (2009). "Consistency Rationing in the Cloud: Pay Only When it Matters." In: *Proceedings of the VLDB Endowment* 2.1, pp. 253–264 (cit. on p. 89).

Kraska, Tim, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete (2013). "MDCC: Multi-Data Center Consistency." In: *Proc. 8th ACM European Conference on Computer Systems*. ACM, pp. 113–126 (cit. on p. 89).

Kung, Hsiang-Tsung and John T Robinson (1981). "On optimistic methods for concurrency control." In: *ACM Transactions on Database Systems (TODS)* 6.2, pp. 213–226 (cit. on p. 74).

Kuper, Lindsey and Ryan R Newton (2013). "LVars: Lattice-Based Data Structures for Deterministic Parallelism." In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pp. 71–84 (cit. on pp. 142, 152, 179, 273).

Lakshman, Avinash and Prashant Malik (2010). "Cassandra: a decentralized structured storage system." In: *ACM SIGOPS Operating Systems Review* 44.2, pp. 35–40 (cit. on pp. 6, 20, 33, 89).

Lamport, Leslie (May 1998). "The Part-Time Parliament." In: *ACM Trans. on Computer Systems* 16.2, pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229 (cit. on pp. 139, 141).

Lamport, Leslie, Sharon Perl, and William Weihl (Dec. 2000). "When Does a Correct Mutual Exclusion Algorithm Guarantee Mutal Exclusion?" In: *Inf. Process. Lett.* 76.3, pp. 131–134. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(00)00132-0. URL: http://dx.doi.org/10.1016/S0020-0190(00)00132-0 (cit. on p. 3).

Lampson, Butler (2001). "The ABCD's of Paxos." In: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*. PODC '01. Newport, Rhode Island, USA: ACM, pp. 13–. ISBN: 1-58113-383-9. DOI: 10.1145/383962.383969. URL: http://doi.acm.org/10.1145/383962.383969 (cit. on p. 139).

Li, Cheng, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis (2014). "Automating the Choice of Consistency Levels in Replicated Systems." In: *USENIX Annual Technical Conference* (cit. on p. 88).

Li, Cheng, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues (2012). "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary." In: *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (cit. on pp. 88, 89).

Li, Jialin, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports (2016). "Just say NO to Paxos overhead: Replacing consensus with network ordering." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA* (cit. on pp. 13, 141).

LibPaxos: Open-source Paxos (n.d.). http://libpaxos.sourceforge.net/. URL: http://libpaxos.sourceforge.net/ (cit. on p. 140).

Liu, Jed, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C Myers (Oct. 2009). "Fabric: A Platform For Secure Distributed Computation and Storage." In: *22nd ACM Symp. on Operating System Principles (SOSP)*, pp. 321–334. URL: http://www.cs.cornell.edu/andru/papers/fabric-sosp09.html (cit. on p. 181).

Liu, Jed and Andrew C Myers (Apr. 2014). "Defining and Enforcing Referential Security." In: *3rd Conf. on Principles of Security and Trust (POST)*, pp. 199–219. DOI: 10.1007/978-3-642-54792-8_11. URL: http://www.cs.cornell.edu/andru/papers/persist (cit. on p. 51).

Lloyd, Wyatt, Michael J Freedman, Michael Kaminsky, and David G Andersen (2011). "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." In: *23rd ACM Symp. on Operating System Principles (SOSP)* (cit. on pp. 6, 36, 76).

Lloyd, Wyatt, Michael J Freedman, Michael Kaminsky, and David G Andersen (2013). "Stronger Semantics for Low-Latency Geo-Replicated Storage." In: *10th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pp. 313–328 (cit. on pp. 76, 88, 90).

Lockerman, Joshua, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan (2018). "The Fuzzylog: A Partially Ordered Shared Log." In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, pp. 357–372. ISBN: 9781931971478 (cit. on p. 21).

Magrino, Tom, Jed Liu, Nate Foster, Johannes Gehrke, and Andrew C Myers (Mar. 2019). "Efficient, Consistent Distributed Computation with

Predictive Treaties." In: *ACM SIGOPS/EuroSys European Conference on Computer Systems* (cit. on pp. 155, 180).

Mahajan, Prince, Lorenzo Alvisi, Mike Dahlin, et al. (2011). "Consistency, availability, and convergence." In: *University of Texas at Austin Tech Report* 11, p. 158 (cit. on p. 21).

Marandi, Parisa Jalili, Samuel Benz, Fernando Pedonea, and Kenneth P. Birman (2014). "The Performance of Paxos in the Cloud." In: *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. SRDS '14. Washington, DC, USA: IEEE Computer Society, pp. 41–50. ISBN: 978-1-4799-5584-8. DOI: 10.1109/SRDS.2014.15. URL: http://dx.doi.org/10.1109/SRDS.2014.15 (cit. on pp. 131, 139).

Matsakis, Nicholas D and Felix S Klock (2014). "The rust language." In: *ACM SIGAda Ada Letters* 34.3, pp. 103–104 (cit. on pp. 262, 264).

Mazieres, D. (2007). "Paxos Made Practical." In: *Technical report. http://www.scs. stanford. edu/dm/home/papers* (cit. on p. 140).

McCullough, Daryl (May 1987). "Specifications for Multi-Level Security and a Hook-up Property." In: *IEEE Symp. on Security and Privacy*. IEEE Press (cit. on p. 50).

Mehdi, Syed Akbar, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd (2017). "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades." In: *NSDI*, pp. 453–468 (cit. on p. 88).

Meiklejohn, Christopher and Peter Van Roy (2015). "Lasp, a language for distributed, coordination-free programming." en. In: *Int'l Symp. on Principles and Practice of Declarative Programming*, pp. 184–195. ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790525. URL: http://dl.acm.

org/citation.cfm?doid=2790449.2790525 (cit. on pp. 24, 142, 152, 179, 273).

Meyers, Scott (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++ 11 and C++ 14*. " O'Reilly Media, Inc." (cit. on p. 77).

Milano, Matthew and Andrew C Myers (June 2018). "MixT: a language for mixing consistency in geodistributed transactions." In: *39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 226–241 (cit. on p. 20).

Minsky, Naftaly H. (1996). "Towards alias-free pointers." In: *ECOOP '96 — Object-Oriented Programming*. Ed. by Pierre Cointe. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 189–209. ISBN: 978-3-540-68570-8 (cit. on pp. 188, 258).

Mitchell, Christopher, Yifeng Geng, and Jinyang Li (2013). "Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store." In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, pp. 103–114. URL: http://dl.acm.org/citation.cfm?id=2535461.2535475 (cit. on p. 143).

Müller, Peter and Arnd Poetzsch-Heffter (1999). "Universes: A type system for controlling representation exposure." In: *Programming Languages and Fundamentals of Programming*. Vol. 263, p. 204 (cit. on p. 259).

Müller, Peter and Arsenii Rudich (2007). "Ownership Transfer in Universe Types." In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 461–478. ISBN: 9781595937865. DOI: 10.1145/1297027.1297061. URL: https://doi.org/10.1145/1297027.1297061 (cit. on p. 259).

Naden, Karl, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff (2012). "A Type System for Borrowing Permissions." In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA, pp. 557–570. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103722. URL: http://doi.acm.org/10.1145/2103656.2103722 (cit. on p. 188).

Nystrom, Nathaniel, Michael R. Clarkson, and Andrew C Myers (Apr. 2003). "Polyglot: An Extensible Compiler Framework for Java." In: *12th Int'l Conf. on Compiler Construction (CC'03)*. Warsaw, Poland: Springer-Verlag, pp. 138–152. ISBN: 3-540-00904-3. DOI: 10.1007/3-540-36579-6_11. URL: http://dx.doi.org/10.1007/3-540-36579-6_11 (cit. on p. 160).

Odersky, Martin (1992). "Observers for linear types." In: *ESOP '92*. Ed. by Bernd Krieg-Brückner. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 390–407. ISBN: 978-3-540-46803-5 (cit. on p. 261).

Ongaro, Diego and John Ousterhout (2014). "In Search of an Understandable Consensus Algorithm." In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro (cit. on p. 141).

Ousterhout, John et al. (July 2011). "The Case for RAMCloud." In: *Commun. ACM* 54.7, pp. 121–130. ISSN: 0001-0782. DOI: 10.1145/1965724.1965751. URL: http://doi.acm.org/10.1145/1965724.1965751 (cit. on p. 141).

Padon, Oded, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham (2016). "Ivy: Safety Verification by Interactive Generalization." In: PLDI '16, pp. 614–630. DOI: 10.1145/2908080.2908118. URL: https://doi.org/10.1145/2908080.2908118 (cit. on p. 121).

Pang, Gene, Tim Kraska, Michael J. Franklin, and Alan Fekete (2014). "PLANET: making progress with commit processing in unpredictable environments." en. In: pp. 3–14. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2588558. URL: http://dl.acm.org/citation.cfm?doid=2588555.2588558 (cit. on p. 179).

Papadimitriou, C. H. (Oct. 1979). "The Serializability of Concurrent Database Updates." In: *Journal of the ACM* 26.4, pp. 631–653 (cit. on p. 37).

Pease, M., R. Shostak, and L. Lamport (Apr. 1980). "Reaching Agreement in the Presence of Faults." In: *J. ACM* 27.2, pp. 228–234. ISSN: 0004-5411. DOI: 10.1145/322186.322188. URL: http://doi.acm.org/10.1145/322186.322188 (cit. on p. 2).

Peter, Simon, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe (Oct. 2014). "Arrakis: The Operating System is the Control Plane." In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, pp. 1–16. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter (cit. on p. 105).

Pierce, Benjamin C. (Feb. 2002). *Types and Programming Languages (The MIT Press)*. The MIT Press. ISBN: 0262162091. URL: https://www.xarg.org/ref/a/0262162091/ (cit. on p. 149).

Plugge, Eelco, Peter Membrey, and Tim Hawkins (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress (cit. on pp. 6, 20, 33, 76, 89).

Poke, Marius and Torsten Hoefler (2015). "DARE: High-Performance State Machine Replication on RDMA Networks." In: *Proceedings of the 24th*

International Symposium on High-Performance Parallel and Distributed Computing. HPDC '15. Portland, Oregon, USA: ACM, pp. 107–118. ISBN: 978-1-4503-3550-8. DOI: 10.1145/2749246.2749267. URL: http://doi.acm.org/10.1145/2749246.2749267 (cit. on p. 141).

Preguiça, Nuno, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos (2003). "Reservations for Conflict Avoidance in a Mobile Database System." In: *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. MobiSys '03. San Francisco, California, pp. 43–56. DOI: 10.1145/1066116.1189038. URL: http://doi.acm.org/10.1145/1066116.1189038 (cit. on pp. 180, 181).

Prisco, Roberto De, Butler W. Lampson, and Nancy A. Lynch (1997). "Revisiting the Paxos Algorithm." In: *Proceedings of the 11th International Workshop on Distributed Algorithms*. WDAG '97. London, UK, UK: Springer-Verlag, pp. 111–125. ISBN: 3-540-63575-0. URL: http://dl.acm.org/citation.cfm?id=645954.675657 (cit. on p. 140).

Protic, Jelica, Milo Tomasevic, and Veljko Milutinovic (1996). "Distributed Shared Memory: Concepts and Systems." In: *IEEE Parallel & Distributed Technology: Systems & Applications* 4.2, pp. 63–71 (cit. on p. 70).

Protzenko, Jonathan et al. (Aug. 2017). "Verified Low-Level Programming Embedded in F*." In: *Proc. ACM Program. Lang.* 1.ICFP. DOI: 10.1145/3110261. URL: https://doi.org/10.1145/3110261 (cit. on p. 188).

RDMA-Paxos: Open-source Paxos (n.d.). https://github.com/wangchenghku/RDMA-PAXOS. URL: https://github.com/wangchenghku/RDMA-PAXOS (cit. on p. 141).

Reed, Eric (2015). "Patina: A formalization of the Rust programming language." In: *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (cit. on p. 264).

Renesse, Robbert van and Fred B. Schneider (2004). "Chain Replication for Supporting High Throughput and Availability." In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI'04. Berkeley, CA, USA: USENIX Association, pp. 7–7. URL: http://dl.acm.org/citation.cfm?id=1251254.1251261 (cit. on p. 140).

Roy, Sudip, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke (2015). "The homeostasis protocol: Avoiding transaction coordination through program analysis." In: *ACM SIGMOD Int'l Conf. on Management of Data*, pp. 1311–1326 (cit. on pp. 155, 180).

Rust Programming Language (2014). http://doc.rust-lang.org/0.11.0/rust.html. URL: http://doc.rust-lang.org/0.11.0/rust.html (cit. on pp. 26, 188).

Sabelfeld, Andrei and Andrew C Myers (Jan. 2003). "Language-Based Information-Flow Security." In: *IEEE Journal on Selected Areas in Communications* 21.1, pp. 5–19. URL: http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf (cit. on pp. 8, 35, 50, 51, 57).

Sabry, Amr and Matthias Felleisen (Nov. 1993). "Reasoning About Programs in Continuation-Passing Style." In: *Lisp and Symbolic Computation* 6.3–4, pp. 289–360. ISSN: 0892-4635. DOI: 10.1007/BF01019462. URL: http://dx.doi.org/10.1007/BF01019462 (cit. on p. 55).

Schneider, Fred B. (Dec. 1990). "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial." In: *ACM Comput. Surv.* 22.4, pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: http://doi.acm.org/10.1145/98163.98167 (cit. on p. 94).

Schönig, Hans-Jürgen (2015). *PostgreSQL Replication*. Packt Publishing Ltd (cit. on p. 179).

Shankland, Steve (May 2008). "Google's Jeff Dean Spotlights Data Center Inner Workings." In: *C|Net Reviews* (cit. on p. 101).

Shapiro, Marc (2017). "A Comprehensive Study of Convergent and Commutative Replicated Data Types." en. In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York, NY: Springer New York, pp. 1–5. ISBN: 978-1-4899-7993-3. DOI: 10.1007/978-1-4899-7993-3_80813-1. URL: http://link.springer.com/10.1007/978-1-4899-7993-3_80813-1 (cit. on pp. 19, 161, 179).

Shasha, Dennis, Francois Llirbat, Eric Simon, and Patrick Valduriez (Sept. 1995). "Transaction chopping: algorithms and performance studies." In: *ACM Trans. on Database Systems* 20.3, pp. 325–363 (cit. on p. 89).

Shudo, Kazuyuki and Takashi Yaguchi (2017). "Causal Consistency for Data Stores and Applications as They are." In: *Journal of Information Processing* 25, pp. 775–782 (cit. on pp. 88, 90).

Sivaramakrishnan, Krishnamoorthy C, Gowtham Kaki, and Suresh Jagannathan (2015). "Declarative programming over eventually consistent data stores." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 50. PLDI '15 6. Portland, OR, USA: Association for Computing Machinery, pp. 413–424. ISBN: 9781450334686. DOI: 10.1145/2737924.2737981. URL: https://doi.org/10.1145/2737924.2737981 (cit. on pp. 33, 38, 87, 88, 90, 179).

Smetsers, Sjaak, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer (1994). "Guaranteeing safe destructive updates through a type system with uniqueness information for graphs." In: *Graph Transformations in Computer Science*. Ed. by Hans Jürgen Schneider and Hartmut Ehrig.

Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 358–379. ISBN: 978-3-540-48333-5 (cit. on p. 261).

Smith, Geoffrey and Dennis Volpano (Jan. 1998). "Secure Information Flow in a Multi-Threaded Imperative Language." In: *25th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 355–364. URL: http://dl.acm.org/citation.cfm?id=268975 (cit. on p. 51).

Stonebraker, Michael and Ariel Weisberg (2013). "The VoltDB Main Memory DBMS." In: *IEEE Data Eng. Bull.* 36.2, pp. 21–27 (cit. on p. 77).

Terry, Doug (Dec. 2013). "Replicated Data Consistency Explained Through Baseball." In: *Commun. ACM* 56.12, pp. 82–89. ISSN: 0001-0782. DOI: 10.1145/2500500. URL: http://doi.acm.org/10.1145/2500500 (cit. on p. 22).

Terry, Douglas B., Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh (2013). "Consistency-based service level agreements for cloud storage." In: *24th ACM Symp. on Operating System Principles (SOSP)* (cit. on p. 89).

Terry, Douglas B., Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer (Dec. 1995). "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." In: *15th ACM Symp. on Operating System Principles (SOSP)*. Copper Mountain Resort, CO, pp. 172–183 (cit. on pp. 37, 179).

Tofte, Mads, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen (2001). *Programming with regions in the ML Kit (for version 4)*. Tech. rep. Citeseer (cit. on p. 262).

Tofte, Mads and Jean-Pierre Talpin (1994). "Implementation of the Typed Call-by-Value λ-Calculus Using a Stack of Regions." In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages. POPL '94. Portland, Oregon, USA: Association for Computing Machinery, pp. 188–201. ISBN: 0897916360. DOI: 10.1145/174675.177855. URL: https://doi.org/10.1145/174675.177855 (cit. on pp. 27, 28, 189, 257, 261, 262).

Tofte, Mads and Jean-Pierre Talpin (1997). "Region-Based Memory Management." In: *Information and Computation* 132.2, pp. 109–176. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1996.2613. URL: http://www.sciencedirect.com/science/article/pii/S0890540196926139 (cit. on p. 262).

Van Renesse, Robbert and Deniz Altinbuken (Feb. 2015). "Paxos Made Moderately Complex." In: *ACM Comput. Surv.* 47.3, 42:1–42:36. ISSN: 0360-0300. DOI: 10.1145/2673577. URL: http://doi.acm.org/10.1145/2673577 (cit. on p. 140).

Vasconcelos, Vasco T. (2012). "Fundamentals of session types." In: *Information and Computation* 217, pp. 52–70. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2012.05.002. URL: http://www.sciencedirect.com/science/article/pii/S0890540112001022 (cit. on p. 261).

Viotti, Paolo and Marko Vukolić (June 2016). "Consistency in Non-Transactional Distributed Storage Systems." In: *ACM Comput. Surv.* 49.1. ISSN: 0360-0300. DOI: 10.1145/2926965. URL: https://doi.org/10.1145/2926965 (cit. on pp. 9, 49).

Vollmer, Michael, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton (2019). "LoCal: A Language for Programs Operating on Serialized Data." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, pp. 48–

62. ISBN: 9781450367127. DOI: 10.1145/3314221.3314631. URL: https://doi.org/10.1145/3314221.3314631 (cit. on p. 262).

Vsync reliable multicast library (Nov. 2011). http://vsync.codeplex.com/ (cit. on p. 141).

Wadler, Philip (1990). "Linear types can change the world!" In: *Programming Concepts and Methods*. Ed. by M. Broy and C. Jones. North Holland (cit. on pp. 27, 261, 264).

Walker, David and Kevin Watkins (Oct. 2001). "On Regions and Linear Types (Extended Abstract)." In: *SIGPLAN Not.* 36.10, pp. 181–192. ISSN: 0362-1340. DOI: 10.1145/507669.507658. URL: https://doi.org/10.1145/507669.507658 (cit. on pp. 261, 263).

Wang, Cheng, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui (Sept. 2017). "APUS: Fast and Scalable Paxos on RDMA." In: *Proceedings of the Eighth ACM Symposium on Cloud Computing*. SoCC '17. Santa Clara, CA, USA: ACM. URL: http://www.cs.hku.hk/research/techreps/document/TR-2017-03.pdf (cit. on p. 140).

Wei, Michael et al. (2017). "vCorfu: A Cloud-Scale Object Store on a Shared Log." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, pp. 35–49. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael (cit. on p. 140).

Wei, Xingda, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen (2015). "Fast In-memory Transaction Processing Using RDMA and HTM." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, pp. 87–104. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815419. URL: http://doi.acm.org/10.1145/2815400.2815419 (cit. on p. 143).

Weiss, Aaron, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed (2019). "Oxide: The essence of rust." In: *arXiv preprint arXiv:1903.00982* (cit. on p. 264).

Whittaker, Michael and Joseph M Hellerstein (2018). "Interactive checks for coordination avoidance." In: *Proceedings of the VLDB Endowment* 12.1, pp. 14–27 (cit. on pp. 155, 180).

Xie, Chao, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan (2014). "Salt: Combining ACID and BASE in a distributed database." In: *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. Vol. 14, pp. 495–509 (cit. on pp. 34, 38, 89).

Xie, Chao, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang (2015). "High-performance ACID via Modular Concurrency Control." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, pp. 279–294. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815430. URL: http://doi.acm.org/10.1145/2815400.2815430 (cit. on p. 89).

Yang, Yingyi, Yi You, and Bochuan Gu (2017). "A Hierarchical Framework with Consistency Trade-off Strategies for Big Data Management." In: *Computational Science and Engineering (CSE) and Embedded and Ubiquitous Computing (EUC), 2017 IEEE International Conference on*. Vol. 1. IEEE, pp. 183–190 (cit. on pp. 34, 89).

Yu, Haifeng and Amin Vahdat (2000). "Design and Evaluation of a Continuous Consistency Model for Replicated Services." In: *4th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (cit. on p. 34).

Zdancewic, Steve and Andrew C Myers (June 2003). "Observational Determinism for Concurrent Program Security." In: *16th IEEE Computer Security Foundations Workshop (CSFW)*. Pacific Grove, California, pp. 29–43. URL: http://www.cs.cornell.edu/andru/papers/csfw03.pdf (cit. on p. 51).

Zdancewic, Steve, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers (Aug. 2002). "Secure Program Partitioning." In: *ACM Trans. on Computer Systems* 20.3, pp. 283–328. URL: http://www.cs.cornell.edu/andru/papers/sosp01/spp-tr.pdf (cit. on p. 53).