A Flexible Type System for Fearless Concurrency

Mae Milano
University of California, Berkeley
Berkeley, CA, USA
mpmilano@berkeley.edu

Joshua Turcotti
University of California, Berkeley
Berkeley, CA, USA
jturcotti@berkeley.edu

Andrew C. Myers
Cornell University
Ithaca, NY, USA
andru@cs.cornell.edu

Abstract

This paper proposes a new type system for concurrent programs, allowing threads to exchange complex object graphs without risking destructive data races. While this goal is shared by a rich history of past work, existing solutions either rely on strictly enforced heap invariants that prohibit natural programming patterns or demand pervasive annotations even for simple programming tasks. As a result, past systems cannot express intuitively simple code without unnatural rewrites or substantial annotation burdens. Our work avoids these pitfalls through a novel type system that provides sound reasoning about separation in the heap while remaining flexible enough to support a wide range of desirable heap manipulations. This new sweet spot is attained by enforcing a heap domination invariant similarly to prior work, but tempering it by allowing complex exceptions that add little annotation burden. Our results include: (1) code examples showing that common data structure manipulations which are difficult or impossible to express in prior work are natural and direct in our system, (2) a formal proof of correctness demonstrating that well-typed programs cannot encounter destructive data races at run time, and (3) an efficient type checker implemented in Gallina and OCaml.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**; **Concurrent programming structures**.

Keywords: concurrency, type systems, aliasing

ACM Reference Format:
Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A Flexible Type System for Fearless Concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523443

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523443

1 Introduction

The promise of a language with lightweight, safe concurrency has long been attractive. Such a language would statically ensure freedom from destructive races, avoiding the cost of synchronization except when concurrent threads explicitly communicate. Our goal is to obtain this "fearless concurrency" [35] for a language with pervasive mutability at its core. Broadly speaking, past efforts to design such a language fall into three camps. Some, like Rust [36], simplify reasoning by severely limiting the shape of representable data structures—making the implementation of common data structures, like the doubly linked list, unapproachable by non-experts[1]. In others [17, 26, 28, 29, 33, 46], harsh limitations on aliasing cause data structure traversal and manipulation to involve significant mutation of the object graph even for simple computations—for example, in these systems removing the tail of a recursively linear singly linked list incurs a write to each list node traversed. Existing approaches that avoid either pitfall require significant programmer annotation to explain aliasing information directly to the compiler [8, 12, 13].

This paper introduces a new type system for fearless concurrency. As in prior work, the goal is to statically ensure that at any point during execution, the part of the heap accessible to a given thread—what we call its *reservation*—is disjoint from the reservations of all other threads. Inspired by Tofte and Talpin [49], the object graph is partitioned into a set of *regions*, a purely compile-time construct which groups objects that enter or leave a thread's reservation as a unit. Neither regions nor reservations are fixed; both can and should change during program execution to reflect the movement of objects among threads. As in prior work [17, 26, 28], our type system supports both inter- and intra-region references; intra-region references may freely link objects within the same region, allowing programmers to easily form arbitrary object graphs, while inter-region references are tracked by the type system and stored in appropriately annotated *isolated* fields. By tracking this information, the type system ensures that threads do not reference objects outside their reservations. Unlike in prior work, this guarantee is provided without requiring that isolated field references satisfy a global domination invariant at all times—and without requiring any annotations from the programmer except at function boundaries.

[1]That doubly linked lists pose a real challenge is affirmed by top search results for "how to write a doubly linked list in Rust" [18, 41].

For rich object graphs, this increased expressive power poses a challenge: to soundly approximate reservations at run time, the type system must accurately determine to which region each accessed object belongs, and further, which regions are contained within the reservation at run time. This determination is made particularly difficult because reservations can grow and shrink dynamically as threads exchange portions of the object graph.

Our key insight begins by leveraging *domination* properties in the heap to force isolated field references to dominate [43] their reachable subgraphs, yielding a notion of encapsulation similar to prior work [29]. We then temper this strong and restrictive global domination property with a new *focus* mechanism inspired by Vault [23]: objects may become temporarily focused, causing their isolated fields' targets to be explicitly tracked by the type system, and thereby *exempted from domination requirements*. This weaker heap invariant, which we call *tempered domination*, allows greater flexibility with lower annotation overhead than in any prior language. It improves on traditional affine-reference languages by enforcing a tree of *regions* rather than a tree of *objects*, allowing more natural structures than are possible in Rust [36]. On the other hand, the focus mechanism skirts the need to maintain a global domination invariant at all times, avoiding the destructive read or swap primitives needed in existing tree-of-regions languages such as L42, LaCasa, Mezzo, and others [3, 4, 17, 26, 28, 46].

Two more novel features enhance expressiveness of our language: (1) a new primitive **if disconnected** that dynamically determines if a region can be safely split at run time, and (2) expressive function types whose parameters and results need not be dominators.

Our type system can naturally represent many mutable data structures found in prior work, without relying on heavy annotations, unnatural representations, destructive reads, or swap primitives. For example, our type system admits straightforward representations of both doubly linked lists with shared ownership and singly linked lists with recursively linear ownership, improving on a motivating example for much prior work [17, 26, 28] in the first case and offering the celebrated mechanisms of uniqueness and borrowing popularized by Rust [36] in the second.

This work brings together the benefits of two traditional lines of prior work without adding significant complexity. For example, both singly and doubly linked lists support traversal, removal, and insertion functions which look much as they would in an introductory programming class, requiring little annotation or run-time overhead. All these operations enjoy fearless concurrency: added elements may have been received from remote threads and removed elements may be immediately sent to a new thread, all without additional dynamic concurrency control mechanisms or the risk of destructive races. No existing language with fearless concurrency can as naturally express this range of data structures.

```
struct sll_node {
  iso payload : data;
  iso next : sll_node?;
}

struct sll {
  iso hd : sll_node?;
}
```

```
struct dll_node {
  iso payload : data;
  next : dll_node;
  prev : dll_node;
}

struct dll {
  iso hd : dll_node?
}
```

Figure 1. A singly linked list and circular doubly linked list. Fields are not nullable by default; the ? annotation on types indicates that this field stores a "maybe" of the appropriate type, effectively making it nullable. The iso keyword enforces transitive domination.

Our primary contributions are summarized as follows:

- A new invariant, **tempered domination**, which allows statically tracked violations of the traditional global domination invariant with a focus construct [23].
- A **region-based type system** capable of tracking the relationships *between* regions, without requiring annotations or explicit scopes to do so.
- A formal paper **proof of soundness** that shows well typed programs have no destructive data races.
- A new primitive to dynamically discover **detailed region graphs** and expose them to static analysis.
- Expressive **function types** capable of statically describing complex heap manipulations.
- A **type checker** implemented in OCaml, and verified in Coq, capable of checking our most complex examples in seconds.

2 A Tail of Two Lists

We begin by explaining key concepts of the new type system, using two linked list implementations as guiding examples.

2.1 Reservations and Tempered Domination

Our language prevents destructive races by dividing the run-time heap into a set of disjoint *reservations*, one per thread. A thread's reservation is the portion of the heap that it may access at any particular time. By keeping reservations disjoint, and ensuring no thread attempts to access an object outside its reservation, we guarantee freedom from destructive races; in other words, it is *reservation-safe.*

As the program executes and threads exchange objects, reservations must shift accordingly. When a thread *sends* an object to another thread, its reservation must lose access to that object's *reachable subgraph*, which includes the object itself as well as *all objects transitively reachable from it.* Conversely, when a thread *receives* an object, its reservation expands; the thread *gains* access to the object and its reachable subgraph.

```
def remove_tail(n: sll_node) : data? {
    let some(next) = n.next in {
        if (is_none(next.next)) {
            n.next = none;
            some(next.payload)
        } else { remove_tail(next) }
    } else { none }
}
```

Figure 2. Removing the final element of a singly linked list. Note that both the returned result and list remain mutable, and the returned result is *no longer* encapsulated by the linked list, unlike in prior work (e.g., [26, 46]). Note also that this function returns none on lists of size one, as it would be impossible to separate the list from its tail in this case.

The key challenge is ensuring reservation safety at compile time. Consider, for example, a linked list containing some abstract payload type data, used as a messaging queue to communicate with other threads. Two possible definitions of such a list are found in figure 1. While these code examples are simple, they expose two key challenges: the ability to represent cyclic data structures, and the ability to traverse trees of unique references. In order to safely add objects received from other threads to either list, or to remove objects from either list to send to other threads, the compiler must reason about *reachability* and *aliasing*, both between the list nodes and their payloads, and between the list nodes themselves.

To make this reasoning tractable for both the compiler and the programmer, our system relies on *transitively dominating references*: references which lie on all paths from the root of the object graph to all objects transitively reachable from that reference. These references are dominators [43] of entire subgraphs; therefore, a thread which loses access to such a reference, for example by sending it to another thread, also loses access to its reachable subgraph. Hence, marking only this single reference as invalid maintains reservation safety. We use the keyword **iso** ("isolated") to describe fields which contain transitively dominating references, thereby exposing knowledge of domination in the object graph to the type system. Looking back to the example code in figure 1, we see that **iso** appears on the list payloads in both linked list implementations, and that it *also* appears on the list spine itself in the case of the singly linked list, indicating that the only way to initially reach a singly linked list node is from its predecessor.

If all **iso** fields contain transitively dominating references, a property we call *global domination*, then we can safely reason about separation in the heap when accessing such data structures. But global domination is too strong a property to be enforced at all times. For example, consider the code in figure 2, which, given the head node, attempts to remove the final element from a singly linked list, returning a dominating reference. The caller of remove_tail may leverage the separation between the removed node and list parameter to, for example, safely send the removed node to a distinct thread without losing access to the list itself[2].

In implementing this function, this code first attempts to dereference the argument's next field, storing it in the variable next. It then checks if next is the tail of the list, removing it from the list and returning its payload if so. Otherwise, it recursively calls remove_tail on the next element. Note something surprising: this code *violates global domination*! Both the next variable and the list parameter hold references to sll_node's **iso**-declared (hence dominating) next field.

In fact, performing a non-destructive traversal of this list while enforcing global domination over all next fields is impossible; all such traversals will require at least a "cursor" variable pointing at the current position in the list, which will necessarily alias the next pointer of that position's predecessor.

Our language thus does *not* enforce a traditionally strict global domination invariant; rather than forcing references stored in **iso** fields to *always* be transitively dominating, we temper this requirement with a type-level mechanism that *explicitly tracks* the targets of some references, requiring transitive domination for exactly those references in **iso** fields which are *not* explicitly tracked by the type system. We call this weakened property *tempered domination*.

Tempered domination generalizes prior work that relies on global domination [26–28, 46]. Crucially, tracking, and indeed the decision of which references to track, occurs without explicit user instruction—requiring annotations only at function boundaries. When we describe the mechanisms in place for preserving tempered domination in the remainder of this paper, we refer to transitively dominating references as simply *dominating references*.

2.2 Aliasing and Reachable Subgraphs

While an otherwise untracked **iso** field in some object o is guaranteed to contain a dominating reference, it is not in general guaranteed that o itself is uniquely referenced; in fact many aliases of any given object may be accessible at any particular time. When checking an **iso** field dereference, it is therefore necessary to ensure the program has not *already* accessed that same object's **iso** field from some other alias.

For example, consider the circular doubly linked list implementation from figure 1. Figure 3 illustrates two possible instances of this list; note that a list of size 1 is represented by a single list node whose prev and next pointers are self-references.

[2]This is in contrast to existing systems [46], in which similar code would still associate the tail with the list even *after* returning it, forever entwining the fate of the tail with that of the list.

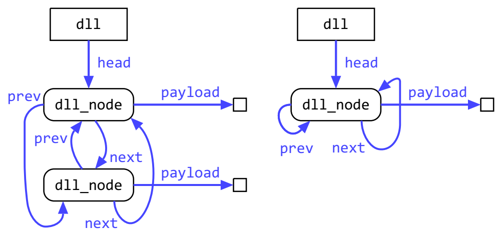

Figure 3. Two circular doubly linked lists, of size 2 and of size 1.

```
def remove_tail(l : dll) : data? {
    let some(hd) = l.hd in {
        let tail = hd.prev;
        tail.prev.next = hd;
        hd.prev = tail.prev;
        some (tail.payload)
    } else { none } }
```

Figure 4. Retrieving the tail of a circular doubly linked list (broken)

As with the singly linked list, we might wish to remove the tail from this circular doubly linked list; our first attempt to do so is in figure 4. This code takes advantage of the circular structure of this list, jumping straight to the end via hd's prev pointer. After patching the list pointers to exclude the tail node, we return the **iso**-annotated `tail.payload` reference. As in the singly linked list, this function has been declared to return only dominating references, so the caller of `remove_tail` should be able to use this payload freely without regard for its former attachment to the list[3].

Sadly, this code contains an error. When passed a list of size 2, this code functions as expected; the tail node is excised from the list, removing all external references to the payload except the one returned from the function itself. But when passed a list of size 1, the code behaves differently: hd and `hd.prev` are in fact *the same object* (fig 3), rendering ineffective the assignments that attempt to remove it from the list. Here, the returned payload actually *isn't* a dominating reference; the list retains the same shape as before, and still provides access to the returned payload. While the programmer could eliminate this error by swapping the payload with a dummy value, that fix is undesirable. It would satisfy the *type checker*, but not remove the bug—replacing a static error with a dynamic one when the dummy value is later unexpectedly encountered.

The correct fix for figure 4 is to add code which handles lists is of length one, perhaps by adding an if-statement. But while this may be sufficient for the *programmer* to know the size of the list a priori, an if-statement alone would not be enough to allow the *type system* to make that same deduction.

[3]This is in contrast to work in the vein of extended Balloon Types [46].

```
def remove_tail(l : dll) : data? {
    let some(hd) = l.hd in {
        let tail = hd.prev;
        tail.prev.next = hd;
        hd.prev = tail.prev;
        // to ensure disjointness for if-disconnected
        tail.next = tail; tail.prev = tail;
        if disconnected(tail,hd) {
            l.hd = some (hd); //l.hd invalid at branch start
            some(tail.payload) }
        else {
            l.hd = none;
            some (hd.payload) }}
    else { none } }
```

Figure 5. Retrieving the tail of a circular doubly linked list (fixed)

To solve this, our work introduces a new primitive conditional form called **`if disconnected`**. This conditional performs a *run-time* check to establish if its arguments' reachable subgraphs are non-intersecting; if they are, it enters the first branch, and otherwise enters the `else` branch. We see this construct in use in figure 5. Here, the existing logic is enhanced by replacing what was once a plain return of `tail.payload` to a call to **`if disconnected`**, returning `tail.payload` when it has been successfully disconnected in size 2+ cases, and returning the head's payload in the size 1 case. Note that the programmer must manually repoint the `tail`'s `next` and `prev` fields away from the remainder of the list, as disconnection is a symmetric property: it is just as essential that `tail` cannot reach head as it is that head cannot reach `tail`. Additionally, the type system does not know which of hd and `tail` connect to `l.hd`, necessitating that `l.hd` be reassigned even in the then branch.

Despite its dynamic nature, the run-time complexity for **`if disconnected`** is quite reasonable—in this example, it would only require reading the metadata of a single object. Notably, the new **`if disconnected`** mechanism cannot be approximated by mechanisms in similar prior work.

3 A Small Language with Dynamic Reservation Safety

We formalize our work as a small core concurrent language with mutable objects, passed by reference.

3.1 Syntax

The syntax of the language can be found in figure 6. Beyond standard imperative constructs, structures, and a first-class "maybe" construct, two novel features stand out: the **`if disconnected`** primitive and blocking messaging primitives **send**-τ and **recv**-τ.

$$
\begin{array}{rl}
\text{(function definition) FDEF} ::= & \texttt{def}\ fn : \tau_{fn}\{e\} \\
\text{(program) } p ::= & \text{FDEF};\ p \mid e \\
\text{(expression) } e ::= & l \mid x \mid e;e \mid e.f \mid e.f = e \mid x = e \mid fn(x,\ldots,x) \mid e \oplus e \mid \texttt{new}\ \tau \mid \texttt{declare}\ x : \tau\ \texttt{in}\ \{e\} \\
& \mid \texttt{if}\ (e)\ \{e\}\ \texttt{else}\ \{e\} \mid \texttt{while}\ (e)\ \{e\} \mid \texttt{send-}\tau(e) \mid \texttt{recv-}\tau() \mid \texttt{if disconnected}(x,x)\ \{e\}\ \texttt{else}\ \{e\} \\
& \mid \texttt{none}\ \tau \mid \texttt{some}(e) \mid \texttt{let some}(x) = (e)\ \texttt{in}\ \{e\}\ \texttt{else}\ \{e\} \\
\text{(evaluation context) } E[] ::= & [];e \mid e.f = [] \mid x = [] \mid [] \oplus e \mid l \oplus [] \mid \texttt{if}([])\{e\}\ \texttt{else}\ \{e\} \\
& \mid \texttt{send-}\tau([]) \mid \texttt{some}([]) \mid \texttt{let some}(x) = ([])\ \texttt{in}\ \{e\}\ \texttt{else}\ \{e\}
\end{array}
$$

Figure 6. Core language syntax

3.2 Semantics

Figure 7 presents selected rules of the small-step semantics for a single thread; explicit concurrency constructs are added in section 7. The only values are locations. The small-step configuration is largely standard, including a store h mapping locations to objects, a stack s mapping variable names to locations, and an expression e which is evaluated with reference to the store and stack.

The final element of the configuration, d, is not standard; this context models the (dynamic) reservation and is consulted whenever a location is used. For example, rules E2 - Variable-Ref-Step and E5a - Final-Reference-Step–Variable check d to confine variable and field reads to locations within the reservation, and E8 - Assign-Var-Step and E7a - Final-Assignment-Step–Variable check d to confine variable and field assignments similarly. If any expression attempts to read or write locations that are *not* in the current reservation, no rules apply and the program cannot step; the program intentionally "gets stuck." By augmenting the small-step semantics with this pervasive dynamic reservation check, we can be guaranteed that—provided reservations are always disjoint—no program can destructively race. In section 4 we introduce a type system for which we have proven progress and preservation (section 6) with respect to this small-step system, in turn proving that no well-typed programs get stuck—and therefore, no reservation checks ever fail. Hence, a real implementation has no need to track the reservation or to perform such checks at run time.

In contrast to the erasable dynamic reservation checks, the **`if disconnected`** mechanism has unavoidable run-time cost. It must ensure that the object graphs reachable from its arguments are non-intersecting, as specified in rules E15a and E15b. A naive implementation of this check would be unacceptably inefficient, as it would require a complete traversal of the object graphs reachable from both arguments; a more efficient implementation is described in section 5.2.

4 Type System

The type system is built around maintaining tempered domination: untracked **`iso`** fields always dominate their reachable subgraph. To establish this invariant, the type system must be able to determine when two different isolated fields may be aliases. For example, in the doubly linked list example from figure 3, the type system must recognize that `hd` and `hd.tail` may be aliases, and so `hd.payload` and `hd.tail.payload` may be as well. It must also ensure that operations which remove an object from the current thread's reservation also render all aliases of this object statically unusable.

4.1 Regions

To track aliasing, the type system uses *regions* [49] to describe disjoint subgraphs of the overall object graph, statically associating each reference with a region in which its target lives. By ensuring that all possible references to the same object are labeled with the same region, the type system can use a set of regions as a conservative compile-time approximation to a run-time reservation. When an object is lost from the reservation, the type system invalidates all references to that object by preventing the use of any references that target its region. Effectively, the type system treats each region as an *affine resource* which is consumed by reservation-shrinking operations on its constituent objects.

For example, figure 8 circles regions in the doubly linked list instances of figure 3. Entire list spines lie in the same region, which causes the static error from in original attempt: both `hd` and `hd.next` are in the same region, so the type system *always* treats them as potential aliases.

4.2 Focus

The tempered domination invariant requires that *untracked* **`iso`** fields must dominate their reachable subgraph, while *tracked* **`iso`** fields are unrestricted. Over the course of program execution, untracked **`iso`** fields may become tracked, and tracked **`iso`** fields may in turn become untracked. To allow tracked **`iso`** fields to be safely untracked, our type system ensures that all tracked **`iso`** fields have statically known target regions. To avoid unsoundness, we must ensure that potential aliases do not have conflicting static tracking information. To this end, we introduce a *focus* mechanism, which allows variables to become tracked only in regions in which no other variables are currently tracked. Since variables from distinct regions are necessarily distinct, this ensures no **`iso`** field ever becomes tracked via multiple aliases. This non-aliasing behavior is formalized as invariant I6 in the appendix.

$$\boxed{(d, h, s, e) \xrightarrow{\text{EVAL}} (d, h, s, e)}$$

$\boxed{\text{E1}}$ - Evaluation-Context-Step

$$\frac{(d, h, s, e) \xrightarrow{\text{EVAL}} (d', h', s', e')}{(d, h, s, E[e]) \xrightarrow{\text{EVAL}} (d', h', s', E[e'])}$$

$\boxed{\text{E2}}$ - Variable-Ref-Step

$$\frac{s(x) = l \qquad l \in d}{(d, h, s, x) \xrightarrow{\text{EVAL}} (d, h, s, l)}$$

$\boxed{\text{E5A}}$ - Final-Reference-Step–Variable

$$\frac{s(x) = l \qquad l, l_f \in d \qquad h_v(l)[f] = l_f}{(d, h, s, x.f) \xrightarrow{\text{EVAL}} (d, h, s, l_f)}$$

$\boxed{\text{E7A}}$ - Final-Assignment-Step–Variable

$$\frac{s(x) = l \qquad l, l_f \in d}{(d, h \uplus (l \mapsto (\tau, v)), s, x.f = l_f) \xrightarrow{\text{EVAL}} (d, h \uplus (l \mapsto (\tau, v[f \mapsto l_f])), s, l_f)}$$

$\boxed{\text{E8}}$ - Assign-Var-Step

$$\frac{l \in d}{(d, h, s \uplus (x \mapsto l_{old}), x = l) \xrightarrow{\text{EVAL}} (d, h, s \uplus (x \mapsto l), l)}$$

$\boxed{\text{E11}}$ - Declare-Var-Step

$$\frac{}{(d, h, s, \texttt{declare } x : \tau \texttt{ in } \{e\}) \xrightarrow{\text{EVAL}} (d, h, s[x \mapsto \bot], e)}$$

$\boxed{\text{E15A}}$ - If-Disconnected-Success-Step

$$\frac{\textit{tracked-set}(r^{\cdot}\langle\rangle; x : r\ \tau; \cdot; h, s) \cap \textit{tracked-set}(r^{\cdot}\langle\rangle; y : r\ \tau; \cdot; h, s) = \emptyset}{(d, h, s, \texttt{if disconnected}(x, y)\ \{e_{succ}\} \texttt{ else } \{e_{fail}\}) \xrightarrow{\text{EVAL}} (d, h, s, e_{succ})}$$

$\boxed{\text{E15B}}$ - If-Disconnected-Failure-Step

$$\frac{\textit{tracked-set}(r^{\cdot}\langle\rangle; x : r\ \tau; \cdot; h; s) \cap \textit{tracked-set}(r^{\cdot}\langle\rangle; y : r\ \tau; \cdot; h; s) \neq \emptyset}{(d, h, s, \texttt{if disconnected}(x, y)\ \{e_{succ}\} \texttt{ else } \{e_{fail}\}) \xrightarrow{\text{EVAL}} (d, h, s, e_{fail})}$$

Figure 7. Selected small-step rules. Full small-step rules can be found in the appendix.

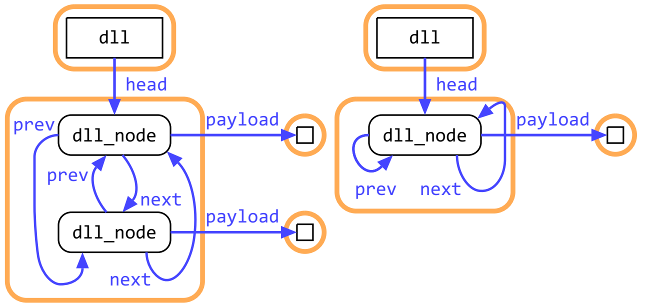

Figure 8. Two circular doubly linked lists, with regions drawn.

$$\begin{array}{rll} \text{(type)} & \tau ::= \textit{Struct} \mid \textit{Struct?} & \mathcal{H} ::= r^{\circ}\langle X\rangle,\ \mathcal{H} \mid \cdot \\ & \circ ::= \dagger \mid \cdot & X ::= x^{\circ}[F],\ X \mid \cdot \\ & \Gamma ::= x : r\ \tau,\ \Gamma \mid \cdot & F ::= f \rightarrowtail r,\ F \mid \cdot \end{array}$$

Figure 9. Surface context definitions for $\mathcal{H}$ and Γ

4.3 Typing Judgments and Static Contexts

The typing judgment has the form $\mathcal{H}; \Gamma \vdash e : r\ \tau \dashv \mathcal{H}; \Gamma$, following the grammar in figure 9. It associates an expression e with a type τ and a *region* r. Recall from section 4.1 that regions are treated linearly in the type system; the transformation of linear contexts is represented not by context splitting [51] but by "input" (before $\vdash$) and "output" (after $\dashv$) contexts. The difference between input and output contexts captures e's effects on the type state. To be well-formed, all static contexts $(\Gamma, \mathcal{H}, X, F)$ cannot contain duplicate bindings for regions, variables, or fields.

The variable typing context Γ is a largely standard binding environment recording the type and region of variables; the *heap context* $\mathcal{H}$ is interpreted as a set of *tracking contexts* of the form $r^{\circ}\langle X\rangle$. Each tracking context begins with a *region capability* r, the complete set of which serves to conservatively approximate the dynamic reservation. Were our tracking contexts to contain *only* this r, they would match the tracking context of LaCasa [28, 29]; indeed, several rules—those which introduce, check, and eliminate regions—require only this level of detail.

4.4 Expression Typing with Tracking Contexts

In addition to the top-level structure describing the set of tracked regions in $\mathcal{H}$, the full tracking context $r^{\circ}\langle x^{\circ}[f \rightarrowtail r, \ldots] \ldots\rangle$ includes a description $x^{\circ}[f \rightarrowtail r, \ldots]$ of the region structure discovered by our focus mechanism: namely, tracked variables x in the region r, where each $f \rightarrowtail r$ maps tracked fields f to their target regions r. Both variables and regions also include a *pinning* annotation described by the metavariable $\circ$. Pinning a region (resp. variable) prevents any new variables (resp. `iso` fields) from becoming tracked in that region (resp. variable). Pinning is necessary when the typing context might only have *partial* static information about the heap, and allows the type system to express abstraction over $\mathcal{H}$.

Figure 10 shows how the context $\mathcal{H}$ is used to type expressions. First, note that $\mathcal{H}$ prevents the type system from confusing an **iso** field with potential aliases; as shown in T5 - Isolated-Field-Reference, no **iso** field of some variable may be accessed unless both that variable and its field are already present in the tracking context, *and* the recorded region targeted by that field is itself present in $\mathcal{H}$.

$$\boxed{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H};\Gamma}$$

T2 - Variable-Ref

$$\frac{x : r\ \tau \in \Gamma \qquad r \in \mathit{regs}(\mathcal{H})}{\mathcal{H};\Gamma \vdash x : r\ \tau \dashv \mathcal{H};\Gamma}$$

T3 - Sequence

$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma' \vdash e' : r'\ \tau' \dashv \mathcal{H}'';\Gamma''}{\mathcal{H};\Gamma \vdash e; e' : r'\ \tau' \dashv \mathcal{H}'';\Gamma''}$$

T4 - Non-Isolated-Field-Reference

$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma' \qquad \cdot\ f\ \tau_f \in \mathit{fields}(\tau)}{\mathcal{H};\Gamma \vdash e.f : r\ \tau_f \dashv \mathcal{H}';\Gamma}$$

T5 - Isolated-Field-Reference

$$\frac{\texttt{iso}\ f\ \tau_f \in \mathit{fields}(\tau) \qquad r^\circ\langle x^{\circ'}[f \rightarrowtail r_f, F], X\rangle \in \mathcal{H} \qquad r_f^{\circ''}\langle X'\rangle \in \mathcal{H}}{\mathcal{H}; x : r\ \tau, \Gamma \vdash x.f : r_f\ \tau_f \dashv \mathcal{H}; x : r\ \tau, \Gamma}$$

T6 - Non-Isolated-Field-Assignment

$$\frac{\mathcal{H};\Gamma \vdash e_f : r\ \tau_f \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma' \vdash e : r\ \tau \dashv \mathcal{H}'';\Gamma'' \qquad \cdot\ f\ \tau_f \in \mathit{fields}(\tau)}{\mathcal{H};\Gamma \vdash e.f = e_f : r\ \tau_f \dashv \mathcal{H}'';\Gamma''}$$

T7 - Isolated-Field-Assignment

$$\frac{\mathcal{H};\Gamma \vdash e_f : r_f\ \tau_f \dashv \mathcal{H}', r^\circ\langle x^{\circ'}[f \rightarrowtail r_{old}, F], X\rangle; x : r\ \tau, \Gamma' \qquad \texttt{iso}\ f\ \tau_f \in \mathit{fields}(\tau)}{\mathcal{H};\Gamma \vdash x.f = e_f : r_f\ \tau_f \dashv \mathcal{H}', r^\circ\langle x^{\circ'}[f \rightarrowtail r_f, F], X\rangle; x : r\ \tau, \Gamma'}$$

T8 - Assign-Var

$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma', x : r_{old}\ \tau \qquad x \notin \mathit{vars}(\mathcal{H}')}{\mathcal{H};\Gamma \vdash x = e : r\ \tau \dashv \mathcal{H}';\Gamma', x : r\ \tau}$$

T10 - New-Loc

$$\mathcal{H};\Gamma \vdash \texttt{new-}\tau : r\ \tau \dashv \mathcal{H}, r^{\cdot}\langle\rangle;\Gamma$$

T11 - Declare-Var

$$\frac{\mathcal{H};\Gamma, x : \bot\ \tau \vdash e : r\ \tau' \dashv \mathcal{H}';\Gamma', x : r_{out}\ \tau \qquad x \notin \mathit{vars}(\mathcal{H}')}{\mathcal{H};\Gamma \vdash \texttt{declare}\ x : \tau\ \texttt{in}\ \{e\} : r\ \tau' \dashv \mathcal{H}';\Gamma'}$$

T13 - If-Statement

$$\frac{\mathcal{H};\Gamma \vdash e_b : r_b\ \texttt{bool} \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma' \vdash e_t : r\ \tau \dashv \mathcal{H}'';\Gamma'' \qquad \mathcal{H}';\Gamma' \vdash e_f : r\ \tau \dashv \mathcal{H}'';\Gamma''}{\mathcal{H};\Gamma \vdash \texttt{if}\ (e_b)\ \{e_t\}\ \texttt{else}\ \{e_f\} : r\ \tau \dashv \mathcal{H}'';\Gamma''}$$

T14 - While-Loop

$$\frac{\mathcal{H};\Gamma \vdash e_b : r_b\ \texttt{bool} \dashv \mathcal{H};\Gamma \qquad \mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H};\Gamma}{\mathcal{H};\Gamma \vdash \texttt{while}\ (e_b)\ \{e\} : r_u\ \texttt{unit} \dashv r_u^{\cdot}\langle\rangle, \mathcal{H};\Gamma}$$

T15 - If-Disconnected

$$\frac{r_x^{\cdot}\langle\rangle, r_y^{\cdot}\langle\rangle, \mathcal{H}; x : r_x\ \tau_x, y : r_y\ \tau_y, \Gamma; \cdot \vdash e_{succ} : r_{out}\ \tau_{out} \dashv \mathcal{H}';\Gamma' \qquad r^{\cdot}\langle\rangle, \mathcal{H}; x : r\ \tau_x, y : r\ \tau_y, \Gamma; \cdot \vdash e_{fail} : r_{out}\ \tau_{out} \dashv \mathcal{H}';\Gamma'}{r^{\cdot}\langle\rangle, \mathcal{H}; x : r\ \tau_x, y : r\ \tau_y, \Gamma \vdash \texttt{if disconnected}\ (x, y)\ \texttt{in}\ \{e_{succ}\}\ \texttt{else}\ \{e_{fail}\} : r_{out}\ \tau_{out} \dashv \mathcal{H}';\Gamma'}$$

T16 - Send

$$\frac{\mathcal{H};\Gamma \vdash e : r_e\ \tau \dashv \mathcal{H}', r_e^{\cdot}\langle\rangle;\Gamma'}{\mathcal{H};\Gamma \vdash \texttt{send-}\tau(e) : r\ \texttt{unit} \dashv \mathcal{H}', r^{\cdot}\langle\rangle;\Gamma'}$$

T17 - Receive

$$\frac{}{\mathcal{H};\Gamma \vdash \texttt{recv-}\tau() : r\ \tau \dashv \mathcal{H}, r^{\cdot}\langle\rangle;\Gamma}$$

TS1 - Virtual-Transformation-Structural

$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma' \qquad (\mathcal{H}';\Gamma') \overset{\text{VIR}}{\rightsquigarrow} (\bar{\mathcal{H}}';\bar{\Gamma}') \qquad r \in \mathit{regs}(\bar{\mathcal{H}}')}{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \bar{\mathcal{H}}';\bar{\Gamma}'}$$

Figure 10. Selected typing rules. Full typing rules can be found in the appendix.

This tracking context also allows **iso** fields to be freely reassigned, even if doing so would create cycles in the object graph. This is safe because tempered domination requires domination only on *untracked* **iso** fields; fields explicitly mentioned in $\mathcal{H}$ are exempt. Consider, for example, type-checking `x.f = e` with T7 - Isolated-Field-Assignment. This rule places *no restrictions* on e beyond ensuring that it type-checks, and that $x.f$ remains valid and tracked after checking e. The rule simply updates $x.f$'s tracking information in the output context.

We sometimes require the tracking context of a region to be empty, containing no tracked variables and thus no tracked fields. As tempered domination weakens global domination only for tracked isolated fields, empty tracking contexts prove that *every* **iso** field within that region contains a dominating reference, and thus is safe to transmit between threads via T16 - Send (which requires an empty context) and T17 - Receive (which assumes one).

Note that rules such as T10 - New-Loc, which add regions, variables, or fields to existing contexts, enforce freshness because well-formed contexts cannot duplicate bindings.

A notable *absence* in figure 10 is any rule which introduces or eliminates elements in a tracking context. This role is played by TS1 - Virtual-Transformation-Structural, which allows invariant-preserving *virtual transformations* to be performed on static contexts.

4.5 Virtual Transformations

Rule TS1 serves to expose a rich language of *virtual transformations* specified by the V rules in figure 11. These rules manipulate $\mathcal{H}$ to match the requirements of the syntax-directed T rules. For example, consider the program $x = \texttt{new-}\tau(); x.f$. After type-checking the first expression in this sequence via T10 and T8 - Assign-Var, we could obtain the following typing judgment:

$$\cdot; x : \bot\ \tau \vdash x = \texttt{new-}\tau() : r\ \tau \dashv r^{\cdot}\langle\rangle; x : r\ \tau$$

If we then moved on to checking $x.f$, rules T3 - Sequence and T5 - Isolated-Field-Reference would seem natural yet be inapplicable. This is because the output context of $\texttt{new-}\tau()$'s derivation has the form $r^{\cdot}\langle\rangle; x : r\ \tau$, but the field reference rule requires a context like $r^{\cdot}\langle x^{\cdot}[f \rightarrowtail r_f]\rangle, r_f^{\cdot}\langle\rangle; x : r\ \tau$.

$$\boxed{(\mathcal{H};\Gamma) \overset{\text{VIR}}{\rightsquigarrow} (\mathcal{H};\Gamma)}$$

$\boxed{\text{V1}}$ - Focus

$$(r^{\cdot}\langle\rangle, \mathcal{H}; x : r\ \tau, \Gamma) \overset{\text{VIR}}{\rightsquigarrow} (r^{\cdot}\langle x^{\cdot}[\,]\rangle, \mathcal{H}; x : r\ \tau, \Gamma)$$

$\boxed{\text{V2}}$ - Unfocus

$$(r^{\circ}\langle x^{\cdot}[\,], X\rangle, \mathcal{H}; \Gamma) \overset{\text{VIR}}{\rightsquigarrow} (r^{\circ}\langle X\rangle, \mathcal{H}; \Gamma)$$

$\boxed{\text{V3}}$ - Explore

$$(r^{\circ}\langle x^{\cdot}[F], X\rangle, \mathcal{H}; \Gamma) \overset{\text{VIR}}{\rightsquigarrow} (r^{\circ}\langle x^{\cdot}[f \rightarrowtail r_f, F], X\rangle, r_f^{\cdot}\langle\rangle, \mathcal{H}; \Gamma)$$

$\boxed{\text{V4}}$ - Retract

$$(r^{\circ}\langle x^{\circ'}[f \rightarrowtail r_f, F], X\rangle, r_f^{\cdot}\langle\rangle, \mathcal{H}; \Gamma) \overset{\text{VIR}}{\rightsquigarrow} (r^{\circ}\langle x^{\circ'}[F], X\rangle, \mathcal{H}; \Gamma)$$

$\boxed{\text{V5}}$ - Attach

$$(r_1^{\cdot}\langle X_1\rangle, r_2^{\circ}\langle X_2\rangle, \mathcal{H}; \Gamma) \overset{\text{VIR}}{\rightsquigarrow} (r_2^{\circ}\langle X_1[r_1 \mapsto r_2], X_2[r_1 \mapsto r_2]\rangle, \mathcal{H}[r_1 \mapsto r_2]; \Gamma[r_1 \mapsto r_2])$$

Figure 11. Virtual Transformation Rules.

Note that these contexts *describe the same heap*! As `x.f` is a dominating reference, it is equally correct to represent it as explicitly tracked or as untracked. This needed shift between different but equivalent representations of the same heap is performed by rule TS1. In this particular case, transformations V1 - Focus and V3 - Explore achieve the desired transformation:

$$(r^{\cdot}\langle\rangle; \_) \overset{\text{VIR}}{\rightsquigarrow} (r^{\cdot}\langle x^{\cdot}[\,]\rangle; \_) \overset{\text{VIR}}{\rightsquigarrow} (r^{\cdot}\langle x^{\cdot}[f \rightarrowtail r_f]\rangle, r_f^{\cdot}\langle\rangle; \_)$$

Note here that V1 - Focus requires the target region to be empty and unpinned, ensuring we do not inadvertently focus two aliases of the same object. Equally, V3 - Explore relies on well-formedness of its contexts to ensure no fields are explored twice. Conversely, the rules V4 - Retract and V2 - Unfocus can be used to transform a heap context in which an explicitly tracked variable points to an empty region into one where both that variable becomes untracked *and* its destination region is dropped, invalidating any other references to the retracted target's region and restoring domination in the process.

4.6 Decidability of Virtual Transformations

An astute reader may note that, unlike the initial typing rules in figure 10, TS1 - Virtual-Transformation-Structural is *not* syntax-directed. We present a decision procedure for type checking with virtual transformations. It runs in common-case polynomial and worst-case exponential time.

In general, given a source $(\mathcal{H}, \Gamma)$ and a target $(\mathcal{H}', \Gamma')$, the problem of discovering a $\overset{\text{VIR}}{\rightsquigarrow}$ path between the two is efficiently decidable by a greedy approach. Effectively, the type checker can defer applying any virtual transformation until it encounters a rule whose type constraints are not satisfied by the current heap context; in the absence of branching constructs, such a deferral can never affect typability. Deciding whether application of TS1 sufficiently transforms typing contexts to allow syntax-directed applications such as T7 - Isolated-Field-Assignment and T16 - Send reduces to this path finding problem.

Unfortunately, *unification* between disparate branches, such as in T13 - If-Statement and T15 - If-Disconnected, cannot rely solely on a greedy approach. To satisfy the conditional typing rules, unification *must* occur at the time of checking the conditional—and there may be many ways to unify its branches which appear equivalent at the time of checking the conditional, but are not equally able to check subsequent expressions. Were we to employ an oracle which can produce a precise target unification context, type checking again becomes efficiently, greedily decidable. In the absence of such an oracle, backtracking search must be performed on the choice of a unification target. We note however that, due to our choice to limit typeable `iso` field accesses to only fields of currently declared variables, the number of $\mathcal{H}$ contexts reachable by virtual transformation is bounded above by the number of variables currently in scope. Thus, even a naive search suffices to obtain completeness, at the cost of run time exponential in the number of variables and the length of the longest function. Heuristics for speeding up search are briefly discussed in section 5.1.

4.7 Abstraction by Framing and Pinning

Figure 12 introduces rule TS2 - Framing-Structural, which exposes our second non-syntax directed typing rule: *framing*. Framing allows our typing rules to ignore irrelevant portions of the static contexts $\mathcal{H}$ and Γ, letting the type checker temporary *frame away* regions in $\mathcal{H}$, variables in Γ, and portions of tracking contexts.

While framing is a standard feature when reasoning about separation [45], its inclusion in our system is complicated by tempered domination. Naively allowing variables within tracking contexts to be framed away would seemingly violate tempered domination; it would take an invariant-satisfying context with explicit domination exceptions, and replace it with one in which no record of those exceptions appears—without making corresponding changes to the heap.

The pinning annotation (4.4) solves this problem. Pinning elements of a tracking context indicates that those elements have *partial information*: that is, it *cannot* be assumed that untracked `iso` fields of a pinned region or variable contain dominating references. By leveraging pinning, we can admit framing rules which weaken elements of tracking contexts without introducing unsoundness. Since a pinned context may *only* be obtained by framing, any pinned context always approximates some fully unpinned context, which avoids the need to further temper tempered domination in our proofs of progress and preservation.

$\boxed{\text{TS2}}$ - Framing-Structural

$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma' \qquad (\mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{A}{\rightsquigarrow}} (\bar{\mathcal{H}};\bar{\Gamma}) \qquad (\mathcal{H}';\Gamma') \overset{\text{FRM}(e)}{\underset{A}{\rightsquigarrow}} (\bar{\mathcal{H}}';\bar{\Gamma}') \qquad r \in regs(\bar{\mathcal{H}}')}{\bar{\mathcal{H}};\bar{\Gamma} \vdash e : r\ \tau \dashv \bar{\mathcal{H}}';\bar{\Gamma}'}$$

$$\boxed{(\mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{A}{\rightsquigarrow}} (\mathcal{H};\Gamma)}$$

$\boxed{\text{F1}}$ - Region-Framing

$$(\mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{\cdot;\bar{\mathcal{H}}}{\rightsquigarrow}} (\mathcal{H} \uplus \bar{\mathcal{H}};\Gamma)$$

$\boxed{\text{F2}}$ - Region-Pinnedness-Framing

$$(r^{\dagger}\langle X\rangle, \mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{r;\cdot}{\rightsquigarrow}} (r^{\cdot}\langle X\rangle, \mathcal{H};\Gamma)$$

$\boxed{\text{F3}}$ - Tracked-Variable-Framing

$$\frac{dom(\bar{X}) \cap (NV(e) \cup dom(\Gamma)) = \emptyset}{(r^{\dagger}\langle X\rangle, \mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{r;\bar{X}}{\rightsquigarrow}} (r^{\dagger}\langle X \uplus \bar{X}\rangle, \mathcal{H};\Gamma)}$$

$\boxed{\text{F4}}$ - Variable-Pinnedness-Framing

$$(r^{\circ}\langle x^{\dagger}[F], X\rangle, \mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{r;x}{\rightsquigarrow}} (r^{\circ}\langle x^{\cdot}[F], X\rangle, \mathcal{H};\Gamma)$$

$\boxed{\text{F5}}$ - Field-Framing

$$(r^{\circ}\langle x^{\dagger}[F], X\rangle, \mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{r;x,\bar{F}}{\rightsquigarrow}} (r^{\circ}\langle x^{\dagger}[F \uplus \bar{F}], X\rangle, \mathcal{H};\Gamma)$$

$\boxed{\text{F6}}$ - Variable-Framing

$$(\mathcal{H};\Gamma) \overset{\text{FRM}(e)}{\underset{\cdot;\bar{\Gamma}}{\rightsquigarrow}} (\mathcal{H};\Gamma \uplus \bar{\Gamma})$$

Figure 12. Framing rules

While TS2 is not syntax-directed, a naive greedy approach forms a sound, complete, and efficient decision procedure for its insertion during type-checking. For details, see the appendix.

4.8 Introducing a Function Abstraction

A function abstraction should capture all available static tracking information about its arguments as input, and allow arbitrary transformations of that information as output. Following this principle, our system provides function types $(\mathcal{H};\Gamma) \Rightarrow (\mathcal{H}';\Gamma';r,\tau)$ with three main components: (1) an input pair $(\mathcal{H},\Gamma)$ in which Γ captures the function's parameters with their expected region and type, and $\mathcal{H}$ captures the tracking contexts of those regions, possibly closed over the tracked isolated references in those contexts; (2) an output pair $(\mathcal{H}',\Gamma')$ which captures the final state of the same variables and regions; and (3) the region r and type τ of the returned value. Rules T0 - Function-Definition and T9 - Function-Application integrate these function types. T0 requires that the function body be well-typed with the given input and output contexts, and T9 requires that, up to renaming of variables and regions, the call site's $\mathcal{H},\Gamma$ match the function's input $\mathcal{H},\Gamma$.

At first glance, this reliance on an exact match of contexts may appear restrictive; however, function declarations need only include elements in $\mathcal{H}$ and Γ relevant to that function's execution. Pinning annotations in the function declaration allow call sites to produce an exact match by using TS2 - Framing-Structural to frame away any irrelevant portions of the application context.

4.9 A Usable Function Syntax

The $\mathcal{H}$ and Γ contexts are complex and would be onerous to expect a programmer to write down directly. We therefore expose an alternate surface syntax for describing function types. This syntax is intended to be more intuitive for programmers, while maintaining the full expressive power of the type system.

Two principles drove the design of this user-facing syntax. The first is that programmers should never directly mention regions, as their direct inclusion in syntax here could lead programmers to expect them to be usable elsewhere in the program. The second is to lean on good defaults that match programmer expectations; only exceptional code should require additional annotation.

Following the principle of good defaults, for unannotated functions, three assumptions hold:

- At input, each parameter comes from a distinct unpinned region with no tracking context.
- At output, each parameter remains in that region, which again must be unpinned and empty.
- A returned result is in its own unpinned, empty region.

These assumptions suffice to write functions that perform in-place manipulations of tree-like isolated data structures. Notably, function requirements are only checked at the beginning and end of each function body; function bodies which only *temporarily* deviate from these expected properties still require no annotation.

In lieu of presenting the full surface language for function declarations, we highlight interesting cases by example in the style of section 2. The `concat` function in figure 14 illustrates an example of the most commonly needed annotation on functions in our system: `consumes`, which indicates the annotated input is *consumed* by the function. A function can consume a parameter in more than one way. Intuitively, it could `send` that parameter to another thread; in the case of figure 14, the parameter is RETRACTED into an **`iso`** field of the other parameter, concatenating the lists together and becoming wholly owned by the larger list in the process. Interestingly, our full implementation of a singly linked list—consisting of 8 functions—requires only this `consumes` annotation, and even then in just two places.

But there is need for function syntax more expressive than just `consumes` annotations. Consider for example the `get_nth_node` function in figure 14. This function takes a circular doubly linked list and returns a mutable reference

$\boxed{\text{T0}}$ - Function-Definition

$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma' \qquad \tau_{fn} = (\mathcal{H};\Gamma) \Rightarrow (\mathcal{H}';\Gamma';r,\tau) \qquad (fn, \tau f) \in \mathcal{F}}{\vdash \mathsf{def}\ fn : \tau f\{e\}}$$

$\boxed{\text{T9}}$ - Function-Application

$$\frac{\overline{\Gamma \vdash x_i : r_i\ \tau_i} \qquad \begin{array}{c}(fn, (\mathcal{H}; x'_1 : r'_1\ \tau_1, \ldots, x'_n : r'_n\ \tau_n) \Rightarrow (\mathcal{H}';\Gamma';r'_0,\tau_0)) \in \mathcal{F} \\ \overline{r'_i \mapsto r_i} \sqsubseteq \Phi_r \in \mathit{bijections}(\mathit{RegionNames}) \qquad \Phi_x = \overline{x'_i \mapsto x_i} \in \mathit{bijections}(\mathit{VariableNames})\end{array}}{\Phi_x(\Phi_r(\mathcal{H}));\Gamma \vdash fn(x_1,\ldots,x_n) : r_0\ \tau_0 \dashv \Phi_x(\Phi_r(\mathcal{H}'));\Phi_x(\Phi_r(\Gamma'))}$$

Figure 13. Function application and definition typing rules

```
def concat(l1, l2 : sll_node) : unit consumes l2 {
 let some(l1_next) = l1.next in {
   concat(l1_next, l2);
 } else { l1.next = some l2;}}

def get_nth_node(l : dll, pos : int) : dll_node?
 after: l.hd ~ result {
   let some(node) = l.hd in {
     while (pos > 0) {
       node = node.next;
       pos = pos - 1
     }; some(node)
   } else { none } }
```

Figure 14. Concatenating two lists, and returning the *n*th node of a doubly linked list

to the *n*th node, wrapping around if necessary. When type-checking an application of this function, it is essential that the type system knows about the relationship between the function's argument and its returned result—namely that, rather than living in its own unrelated region as would be the default, the function result lives in the same region as the argument's **iso** hd field. We capture this relationship with the syntax `after` : $a \sim b$, which means that after the function returns, the regions of objects a and b are the same. Here, a and b could be variables, fields, or the return result itself as in this example. Combined with the pinning syntax, this $\sim$ syntax suffices to regain the full expressive power of function types. Programmers can cleanly express functions—like `get_nth_node`—that would be difficult if not impossible to represent in prior work.

5 Implementation

The type system has been implemented as a prover–verifier architecture which we have made publicly available. The prover is written in ~4,100 lines of OCaml, and its output typing derivations are checked by a verifier written in ~2,000 lines of Coq, making it easy to check by inspection that the type system is implemented faithfully.

5.1 Heuristics for Virtual Transformation Search

As discussed in section 4.6, the TS1 rule in our type system, governing focus, explore, and all other virtual transformations necessary to transform the heap context, is not syntax-directed. Several heuristics implemented by the type checker keep type checking efficient in practice. In particular, we aim to avoid backtracking search when unifying the branches of a conditional.

At the heart of the difficulty in unifying the typing contexts of branches is the information loss associated with key virtual transformations such as V2 - Unfocus and V5 - Attach. Unification can thus be viewed as the problem of inferring which linear resources must be preserved to type-check a given program suffix. By employing liveness analysis of variables and isolated fields as a unification oracle, our checker can verify our largest examples in a handful of seconds. When necessary, our tool still falls back to search. Other approaches—such as user annotations or an external constraint solver—may be useful for pathological cases. More details appear in the appendix.

5.2 Efficiently Checking Mutual Disconnection

We implemented a version of the **if disconnected** check (introduced in section 3.2) that is efficient based on two usage assumptions. The first assumption is that data structure designers prefer to keep regions small when possible, placing the **iso** keyword at abstraction boundaries—for example, collections place their contents in **iso** fields, as we do in figure 1. The second assumption is that **if disconnected** is commonly used to detach a small portion of a region—often as small as a single object (as in figure 5).

Following these assumptions, we propose a two-step process for the efficient implementation of **if disconnected**. First, store a reference count which tracks immediate heap references stored in non-**iso** fields of structures. This stored reference count is updated *only* on field assignment, and does not need to be modified—or checked—on assignment to local variables, function invocation, or at any other time. Thus, it is lighter-weight than conventional reference counts.

Second, the **if disconnected** check itself is implemented via interleaved traversals of the object graphs rooted by its two arguments, ignoring references which point outside the current region, and stopping when the *smaller* of the two has been fully explored (or a point of intersection has been found). During this traversal, the algorithm counts the number of times it has encountered each object, assembling a *traversal reference count*. At the end of the traversal, it compares this traversal reference count with the *stored* reference count, concluding that the object graphs are disconnected if the counts match, and conservatively assuming that they remain connected if the counts do not match.

The soundness of this strategy relies on two things: tempered domination enforced on **iso** fields by the type system, and accuracy of the stored heap reference counts. The typing rule for **if disconnected** ensures that its arguments come from the same region, and that nothing within that region is tracked. Each untracked `iso` field roots a distinct, fully independent object graph; thus no object beyond an `iso` field can be the first point of intersection between **if disconnected**'s arguments. This eliminates any need for the traversal to search beyond an **iso** field.

Our choice to terminate the traversal after only the *smaller* graph is explored, meanwhile, is justified by reference counts. The fear here is that, by terminating our exploration early, we may have missed some path from the larger object graph into the smaller. Such a path would necessarily include an unexplored reference targeting an object in the smaller graph. The existence of this unexplored reference would be reflected in the stored reference count, causing the stored reference count to exceed the traversal reference count.

Can this check be done efficiently? For cases which follow our expected use patterns—like the one in figure 5, where the smaller graph's non-**iso** references point only to the object itself—the traversal terminates immediately after encountering only a single object, or a small number of closely linked objects. But in the worst case, this check may involve traversing an entire region of arbitrary size. Such a traversal would cut against the intended use-cases of **if disconnected**; we would thus consider these uses more likely to arise as a result of buggy code than of intentional design. In these buggy cases, our **if disconnected** check would still improve on systems which rely on destructive reads, replacing unexpected run-time crashes later in the program with a static error (or an unexpectedly slow no-op) at the point the bug actually occurs. Returning to figure 5, even were we to introduce a bug by failing to correctly disconnect the object graph—for example by omitting the assignments which immediately precede the **if disconnected** check—the resulting traversal would incur nearly no additional cost, with **if disconnected**'s check still terminating after only two objects are encountered.

6 Correctness

We have discussed in detail the surface syntax and small-step semantics of our language, whose rules guarantee that any attempt to access a location outside the dynamic reservation d will arrest the program in a "stuck" state, and we have presented typing rules with a complex context $\mathcal{H}$, which statically models capabilities to access a shared heap. The missing piece of the puzzle is a run-time invariant using the information in $\mathcal{H}$ to guarantee that well-typed programs never encounter that stuck state. This is easily phrased:

Definition (Invariant $\boxed{\text{I1}}$ - Reservation-Sufficiency)**.** All locations that could be the result of stepping a well-typed expression are contained in the dynamic reservation d.

An immediate consequence of I1 is that any variables bound in Γ to a region tracked in $\mathcal{H}$ are mapped (by the dynamic stack s) to a location in d. This is because T2 - Variable-Ref guarantees well-typed access to any such variables, and E2 - Variable-Ref-Step steps them directly to their bound locations. Similarly, transitive targets of fields are in d. Invariant I1 is thus exactly the missing piece to bind well-typedness to reservation safety. Naturally, its preservation as programs step is a nontrivial proof goal, so we introduce a second invariant I2 which implies I1 and is closer to the formalisms of the language:

Definition (Invariant $\boxed{\text{I2}}$ - Tree-Of-Untracked-Regions)**.** Any two paths in the dynamic heap that begin in a tracked region and terminate at the same location traverse the same sequence of untracked isolated references.

This invariant is fundamental because it directly encodes the core tempered domination invariant: in particular, that beyond our statically tracked set we can assume that all `iso` fields contain dominating references.

To further motivate I2, recall that the accepted static evidence for the separation of two objects is their presence in separate regions (consider T16 - Send), and that untracked isolated references are always assumed to point to untracked regions (see V3 - Explore). Thus, a necessary condition for safety is that locations serving as the target of untracked isolated references may never be bound to variables in tracked regions; otherwise, that variable could be accessed even after is dropped from the reservation. I2 captures this condition.

The appendix formalizes both I1 and I2, as well as additional formal invariants encoding expected agreement between the static and dynamic contexts. All of these invariants together capture the notion of a *sound* configuration used in the following theorems.

Theorem 6.1 (Progress)**.** *Given the well typed expression* $\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma'$ *with sound configuration* $(\mathcal{H},\Gamma,d,h,s,e)$*, there exists a step* $(d,h,s,e)\xrightarrow{\text{EVAL}}(d',h',s',e')$

Theorem 6.2 (Preservation)**.** *Given the well typed expression* $\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma'$ *with sound configuration* $(\mathcal{H},\Gamma,d,h,s)$ *and step* $(d,h,s,e)\xrightarrow{\text{EVAL}}(d',h',s',e')$*, there exist* $\bar{\mathcal{H}},\bar{\Gamma}$ *such that* $\bar{\mathcal{H}};\bar{\Gamma} \vdash e' : r\ \tau \dashv \mathcal{H}';\Gamma'$ *and the configuration* $(\bar{\mathcal{H}},\bar{\Gamma},d',h',s')$ *is sound.*

Proofs of 6.1 and 6.2 are provided in the appendix. Together, these theorems imply that invariants I1 and I2 hold across the execution of a well typed program. This establishes tempered domination is preserved, and it establishes the core safety property of our system: in a well typed program, no thread accesses memory outside its reservation.

7 Concurrency

The results from section 6 show that our system can guarantee the reservation safety of sequential programs. Importantly, this result also means that concurrency is safe.

We model general, message-passing concurrency through the expressions send-$\tau(e)$ and recv-$\tau()$ (T16 - Send and T17 - Receive in the type system of section 4).

The concurrent configuration consists of a single shared heap h, and an n-tuple of threads, each with its own reservation d_i, variable store s_i and expression e_i currently under evaluation. *Soundness* of a concurrent configuration consists of the respective soundness and well-typedness of each thread's e_i with respect to the configuration (d_i, h, s_i), along with pairwise disjointness of the reservations d_i.

Stepping a concurrent configuration occurs by stepping an individual thread, by updating that thread's d_i, s_i, e_i as well as the shared h, or by stepping two threads together that have reached a send-τ/recv-τ pair. This stepping rule is illustrated in figure 15. It steps in the context of the shared heap h, but only updates the respective reservations and expressions of the sending and receiving threads. In particular, it identifies the location l_{root} that the sending thread has chosen, reads h to identify the set d_{sep} of locations that are *live* (i.e., reachable) from l_{root}, and steps if d_{sep} is entirely contained within the sending thread's reservation, transferring it to the receiving thread's reservation along with access to the location l_{root}.

Progress and Preservation in the concurrent configuration are also stated and proved in the appendix, notably establishing that no thread's soundness relies on h outside of its reservation d_i, and that the rules T16 and T17 are sufficient to conclude EC3 - Communication-Paired-Step can be applied without getting stuck on ownership transfer, yielding sound post-transfer configurations for both threads.

8 Expressiveness

To explore the expressiveness of the type system, we have written thousands of lines of algorithmic code, data structure manipulations, and experimented with function abstractions ranging from trivial to pathological. Large samples of this code are presented in the appendix, including complete singly and doubly linked lists and a red–black tree.

Our experience suggests that functions in our language place no unnatural restrictions on common coding patterns, requiring annotations only when the **iso** keywords are added to struct definitions. Further, even functions that manipulate structs with **iso** fields need no annotation unless they take or return object graphs that violate the tempered domination invariant—for example, overlapping object graphs and non-tree object graphs.

We have found that functions whose arguments' object graphs overlap (like the `get_nth_node` example) are usually easy to annotate, while functions that deviate from tempered domination at function boundaries are improved by signature-level annotations describing the shape of their isolated object graph. As an example, the `shuffle` function of the appendix's red–black tree takes 7 tree nodes in an arbitrary, possibly deeply aliased state and returns them with a fixed, tree pointer structure. Expressing that information in the signature provides a level of static safety usually found only in dependently typed languages.

Thus, our experience suggests that besides offering strong safety guarantees, this language is intuitively usable.

9 Related Work

The type system we propose owes much to the rich history of related language designs. In particular, it exploits innovations from several important lines of research: ownership types and capabilities, regions, and linear types (and linear regions). We now attempt to broadly characterize notable work from each line of research, and discuss how our work differs.

9.1 Ownership Types and Nonlinear Uniqueness

While we use the terminology of focus [23] and regions [48], the closest antecedent to *focus* is in CQual [1, 25], while the closest cousin to our regions is *ownership contexts* [16]. The primary difference between our regions and ownership contexts 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.

Recognizing these limitations, later work introduced the ability to mix ownership with *uniqueness* [2, 3, 8, 31, 38, 46]. These languages all enforce uniqueness strictly: a unique reference is the *only* reference that points to its referent. Clarke and Wrigstad weakened this constraint by introducing the idea of *external* uniqueness, and with it the idea of a *dominating reference*: an externally unique reference is traversed on all paths from roots to the object to which it refers [14, 15]. Externally unique references are similar to **iso** fields, but **iso** fields dominate *all objects reachable* from their target, while—in its original formulation—external references dominate just their target. This weaker invariant prevents externally unique references from implying *transitive* ownership. Other variations on ownership also exist; some work makes owning objects explicit, abstracts them with capabilities, or views them as modifiers [9, 13, 17, 20, 28, 29, 39, 40].

Of particular note is the LaCasa language of Haller and Odersky [28, 29], which our work subsumes. LaCasa's surface language (and accompanying annotation burden) are

$$\boxed{h \vdash (d, e; d, e) \xrightarrow{\text{comm-eval}} (d, e; d, e)}$$

$\boxed{\text{EC3}}$ - Communication-Paired-Step

$$\frac{d_{sep} = \mathit{live\text{-}set}(r^{\cdot}\langle\rangle; \cdot; l : r\ \tau; h; \cdot)}{h \vdash (d_a \uplus d_{sep}, E_a^*[\texttt{send-}\tau(l_{root})]; d_b, E_b^*[\texttt{recv-}\tau()]) \xrightarrow{\text{comm-eval}} (d_a, E_a^*[\texttt{new-unit}]; d_b \uplus d_{sep}, E_b^*[l_{root}])}$$

Figure 15. Stepping `send`/`recv` pairs in the concurrent configuration

quite similar; both designs have **`iso`** (`@unique`) fields that dominate the reachable object graph; both annotate methods similarly and rely on linearly tracked region capabilities.

A major limitation of these systems, including LaCasa, is their inability to change thread reservations without mutating objects. Each system has a way to drop an object from a thread’s reservation, rendering the thread unable to use the object subsequently. Lacking a focus mechanism, these languages cannot determine which references need to be invalidated when an object is lost. Rather than make lost objects statically inaccessible, most employ a “destructive read” that implicitly nulls them instead [2, 5, 7, 8, 15], though other approaches exist, such as “swap” [28, 29, 33]. Other systems, such as L42 and Servetto’s extension to Balloon types [26, 46], have a notion of “lending” a reference, allowing the tail to be returned from a list, but *without* separating it from the list abstraction, and thus also without needing to invalidate potential aliases. These systems cannot efficiently implement the `remove_tail` function from figure 2; to truly free the tail of the list from its original owner, they would require a write operation to each node in the list in order to repair destructive reads performed on the way down. Some systems adopt (or aim to adopt) Alias Burying [10] to avoid implicit nulling when all aliases to a unique object are dead, but this mechanism could *not* repair the linked list example.

9.2 Linear Systems and Regions

Since initially popularized by Wadler [51], many linear languages have been proposed [21, 36, 42, 47, 50, 52] which can prevent destructive races without relying on destructive reads or swapping—but at the cost of making direct representations of graph data structures cumbersome. These languages would not be able to directly represent the doubly linked list from figure 1. Much of the recent interest around this class of languages has centered on Rust [36], the first such language to gain widespread adoption [32, 34, 44, 53]. While Rustaceans have discovered a variety of clever ways to simulate cyclic data structures within its type system, those techniques often resemble how our system would behave were one to have a single object per region; complex graphs are possible, but the cost is a dramatic increase in static tracking, much of it borne directly by the user in the form of extra annotations, a reliance on unsafe code, or “clever hacks” like using indices into a linearly owned array as a stand-in for references. Recent work into using “ghost cells” to achieve cyclic data structure patterns is encouraging, but remains above the annotation budget that we believe is desirable for such common data structures [54].

Tofte and Talpin introduced the idea of regions [19, 30, 48], which enable safe stack-based memory management in a language with dynamic allocation. A hallmark of region-based type systems is that functions types specify the regions the function may access [49]. The largest difference between our regions and those of Tofte and Talpin is that our regions are not fixed. They can be merged, renamed, retracted into and explored out from other regions. This flexibility removes the need for complex effect annotations on functions; we can represent complex object graphs by their simple entry points, and declare functions only as taking these entry points.

Our language tracks regions *linearly*. While most existing work that uses linear regions relies on a “swap” or destructive-read primitive [6, 21, 24, 28, 29], some existing work features the ability to “open” a region and freely access the objects within it for a limited scope—much as our language can temporarily focus objects [23, 52].

Fähndrich and DeLine’s Vault language [23] directly inspired our *focus* mechanism. Vault is a primarily linear language for reasoning about protocol state; its focus allows particular objects to be freely aliased, exempting them from the requirements of linearity. A linear field of a potentially nonlinear object in Vault is roughly analogous to **`iso`** fields in our type system. This analogy is rough, however; our **`iso`** fields may refer to objects that are freely aliased within their region, while Vault’s linear fields must be unique references. As in our work, Vault prevents access to **`iso`** fields unless their containing object is focused, though only for writing; reading is always permitted

In comparison, our system requires less rigid management of focused objects and does not enforce linearity on **`iso`** objects themselves—just on their regions. All references—even those in **`iso`** fields—can point to objects that participate in cycles; this would not be possible in Vault, reducing the ease of implementing the doubly linked list in figure 1. Additionally, adoption and focus in Vault are annotation-heavy; Vault does not infer necessary focus points, so the programmer must explicitly fold and unfold accessible object trees.

A Vault-like adoption mechanism is also found in the Mezzo language [4] to allow non-tree object graphs. It is missing the accompanying `focus`, however, which may pose problems interacting safely with Mezzo's novel take on destructive reads. It is unclear if Mezzo's adoption mechanism allows the formation of arbitrary graphs, or only DAGs, but it is difficult to see how a doubly linked list could be implemented in Mezzo without relying on implicit nulling.

9.3 Immutability and Fractional Permissions

Several related systems offer the ability to temporarily share mutable objects with immutable references, and to recover mutability once all shared references' lifetimes have ended [17, 23, 27]. This banner feature of Rust [36] appears in Mezzo [4] and was added to Vault through Boyland's work on fractional permissions [11]. M# [27], an evolution of Sing# [22], also features recovering mutability—later generalized by Pony [17] and L42 [26].

To determine the lifetime of concurrently shared immutable references, these systems all support mutability recovery only when using *structured parallelism* or explicit *recovery scopes*: all possible aliases, including those passed to threads, will have been reclaimed by a statically known program point—usually when all other threads involved in communication have died. In contrast, our simple, unstructured send and receive mechanism cannot track *which* references are transmitted. Threads have no lifetime, so it is impossible to know if a reference sent to another thread is ever returned.

Instead, we expect to take the approach outlined in Gallifrey [37], in which a dynamic mechanism manages shared immutability *and* mutability by relying on replication. Alternatively we could leverage our equivalent of "lending" references to functions during a function call; making those calls asynchronous, and providing a built-in future mechanism by which "lent" references may be returned, is a promising avenue by which recoverable mutability may be supported.

9.4 Significant Complexity

Several systems manage to ensure reservation safety and avoid implicit null (or swap), but introduce significant user-facing complexity [8, 12, 13, 17, 33]. These languages frequently feature explicit, exact region or ownership annotations, provide a type parameterization mechanism which allows the creation of classes whose ownership or region information is determined at instantiation time, or rely on a multitude of reference qualifiers capable of discussing exactly how various objects may relate in the object graph. While such systems are quite flexible, they force the user to reason directly about concepts, like regions and region membership, which we intentionally keep implicit. Here the complexity does not appear to be incidental; it is not clear how to identify a "simple core" language that would be complete on its own. Indeed, our experience designing this type system speaks to the speed at which complexity can creep in from apparently innocuous design choices.

Table 1. Comparison with related language designs.

Language	**sll**	**dll-repr**	**Simple**
Rust	✓	×	~
Unique	✓	×	~
Vault	✓	~	~
Mezzo	~	~	✓
LaCasa	×	✓	✓
OwnerJ	×	✓	✓
Pony	~	✓	~
M#	×	✓	✓
This paper	✓	✓	✓

9.5 Comparison with Closely Related Work

The systems that come closest to matching our design goals are summarized in table 1. In the "sll" column, systems are marked that can implement `remove_tail` from our singly linked list (without requiring O(list-size) object mutations). The "dll-repr" has a check for systems that can directly represent the doubly linked list at all, and the "simple" column marks systems which require few annotations for straightforward implementations of common list mutations. To the best of our knowledge, no previous system is able to represent `remove_tail` from a doubly linked list without relying on destructive reads or a swap primitive. Finally, the "OwnerJ" row captures the close descendants of original ownership type systems, including PRFJ [8] and AliasJava [2] (section 9.1), while the "Unique" row captures the limitations of type systems in the style of Wadler's popularization [51].

10 Conclusion

We started by observing that expressiveness-limiting heap invariants and intimidating annotations are fatal flaws in existing safe concurrency approaches. Our core insights are that these invariants can be weakened without losing power as long as they stay *recoverable* through virtual transformations, and that careful type-system design can preserve decidability in lieu of annotation. The result is a type system that replaces stricture with flexibility and caution with fearlessness—a new sweet spot in this design space that lowers the cost of safe concurrency and opens promising avenues for future work.

Acknowledgments

We thank the anonymous reviewers, Rolph Recto, Tom Magrino, Gabriel Matute, Justin Lubin, Marco Servetto (author of [46]), and our shepherd, Ralf Jung, for their feedback and suggestions. This work was supported by the National Science Foundation under Grant No. 1717554.

References

[1] Alex Aiken, 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* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 129–140.

[2] Jonathan Aldrich, 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). 311–330.

[3] Paulo Sérgio Almeida. 1997. Balloon Types: Controlling Sharing of State in Data Types. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 32–59.

[4] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 4 (2016), 1–94.

[5] Anindya Banerjee 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* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 166–177.

[6] Nels E. Beckman, 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* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 227–244.

[7] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. *SIGPLAN Not.* 37, 11 (Nov. 2002), 211–230.

[8] Chandrasekhar Boyapati and Martin Rinard. 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.

[9] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27.

[10] John Tang Boyland. 2001. Alias Burying: Unique variables without Destructive Reads. *Software: Practice and Experience* 31, 6 (2001), 533–553.

[11] John Tang Boyland. 2010. Semantics of Fractional Permissions with Nesting. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 22 (Aug. 2010), 33 pages.

[12] John Tang Boyland and William Retert. 2005. Connecting Effects and Uniqueness with Adoption. In *Proceedings of the 32nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 283–295.

[13] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:26.

[14] Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 – Object-Oriented Programming*, Luca Cardelli (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–200.

[15] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *Asian Symposium on Programming Languages and Systems*. Springer, 139–154.

[16] David G. Clarke, 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* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64.

[17] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *5th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*. 1–12.

[18] Russell Cohen. 2018. Why Writing a Linked List in (safe) Rust is So Damned Hard. https://rcoh.me/posts/rust-linked-list-basically-impossible/

[19] Karl Crary, 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* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275.

[20] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. 2007. Universes for Race Safety. *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)* (2007), 20–51.

[21] Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *ECOOP 2004 – Object-Oriented Programming*, Martin Odersky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 465–490.

[22] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. 2006. Language Support for Fast and Reliable Message-Based Communication in Singularity OS. *SIGOPS Oper. Syst. Rev.* 40, 4 (April 2006), 177–190.

[23] Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[24] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *European Symposium on Programming*. Springer, 7–21.

[25] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[26] Paola Giannini, Marco Servetto, and Elena Zucca. 2016. Types for Immutability and Aliasing Control. In *ICTCS*, Vol. 16. 62–74.

[27] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 21–40.

[28] Philipp Haller and Alex Loiko. 2016. LaCasa: Lightweight Affinity and Object Capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 272–291.

[29] Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *European Conference on Object-Oriented Programming*. Springer, 354–378.

[30] Fritz Henglein, 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* (Florence, Italy) *(PPDP '01)*. Association for Computing Machinery, New York, NY, USA, 175–186.

[31] John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, USA) *(OOPSLA '91)*. Association for Computing Machinery, New York, NY, USA, 271–285.

[32] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. Association for Computing Machinery, New York, NY, USA, 13–22.

[33] Trevor Jim, 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*. 275–288.

[34] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages.

[35] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

[36] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[37] Mae Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers. 2019. A Tour of Gallifrey, a Language for Geodistributed Programming. In *3rd Summit on Advances in Programming Languages (SNAPL)*.

[38] Naftaly H. Minsky. 1996. Towards Alias-Free Pointers. In *ECOOP '96 — Object-Oriented Programming*, Pierre Cointe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 189–209.

[39] Peter Müller and Arnd Poetzsch-Heffter. 1999. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, Vol. 263. 204.

[40] Peter Müller 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* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 461–478.

[41] ndrewxie (https://users.rust lang.org/u/ndrewxie). 2019. What's the "best" way to implement a doubly-linked list in Rust? https://users.rust-lang.org/t/whats-the-best-way-to-implement-a-doubly-linked-list-in-rust/27899

[42] Martin Odersky. 1992. Observers for Linear Types. In *ESOP '92*, Bernd Krieg-Brückner (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 390–407.

[43] Reese T. Prosser. 1959. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference* (Boston, Massachusetts) *(IRE-AIEE-ACM '59 (Eastern))*. Association for Computing Machinery, New York, NY, USA, 133–138.

[44] Eric Reed. 2015. Patina: A Formalization of the Rust Programming Language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015).

[45] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

[46] Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. 2013. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*. 107.

[47] Sjaak Smetsers, 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*, Hans Jürgen Schneider and Hartmut Ehrig (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 358–379.

[48] Mads Tofte 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* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 188–201.

[49] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176.

[50] Vasco T. Vasconcelos. 2012. Fundamentals of Session Types. *Information and Computation* 217 (2012), 52–70.

[51] Philip Wadler. 1990. Linear types can change the world!. In *Programming Concepts and Methods*, M. Broy and C. Jones (Eds.). North Holland.

[52] David Walker and Kevin Watkins. 2001. On Regions and Linear Types (Extended Abstract). *SIGPLAN Not.* 36, 10 (Oct. 2001), 181–192.

[53] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *arXiv preprint arXiv:1903.00982* (2019).

[54] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.* 5, ICFP, Article 92 (aug 2021), 30 pages.

MixT: A Language for Mixing Consistency in Geodistributed Transactions

Matthew Milano
Dept. of Computer Science, Cornell University
Ithaca, New York, United States of America
milano@cs.cornell.edu

Andrew C. Myers
Dept. of Computer Science, Cornell University
Ithaca, New York, United States of America
andru@cs.cornell.edu

Abstract

Programming concurrent, distributed systems is hard—especially when these systems mutate shared, persistent state replicated at geographic scale. To enable high availability and scalability, a new class of weakly consistent data stores has become popular. However, some data needs strong consistency. To manipulate both weakly and strongly consistent data in a single transaction, we introduce a new abstraction: mixed-consistency transactions, embodied in a new embedded language, MixT. Programmers explicitly associate consistency models with remote storage sites; each atomic, isolated transaction can access a mixture of data with different consistency models. Compile-time information-flow checking, applied to consistency models, ensures that these models are mixed safely and enables the compiler to automatically partition transactions. New run-time mechanisms ensure that consistency models can also be mixed safely, even when the data used by a transaction resides on separate, mutually unaware stores. Performance measurements show that despite their stronger guarantees, mixed-consistency transactions retain much of the speed of weak consistency, significantly outperforming traditional serializable transactions.

CCS Concepts • **Information systems** → **Distributed database transactions**; • **Software and its engineering** → **Consistency**; **Domain specific languages**;

Keywords Consistency, Transactions, Information Flow

ACM Reference Format:
Matthew Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192375

1 Introduction

Programmers often face the task of writing concurrent, distributed systems that share mutable, persistent state across geographic distances. Traditional 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.

New tools have evolved to fill the gap, relying on weaker consistency to enable lower latencies and higher availability. Evidence from both industry [8, 20, 31, 35, 46] and academia [13, 27, 51] suggests that weak consistency is viable, especially when accompanied by familiar transactional abstractions, clear consistency guarantees, and efficient operations over persistent data.

But problems remain. While weaker consistency models suffice for certain application data, other data needs stronger consistency—single applications might need multiple *levels* of consistency. Running with a single consistency level, as seen in prior systems [7, 15, 29, 56, 58, 59]), can be slow [5]; 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. Based on this observation, we introduce *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 [6]. 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.

Combining these challenges, we present MixT, a domain-specific language for mixed-consistency transactions. In MixT, persistent data and operations at various stores can be accessed with strong guarantees (§3). To ensure the semantic guarantees of mixed-consistency transactions, weaker-consistency information should avoid influencing stronger-consistency information. To prevent this influence, we view consistency as a property of data, treating consistency as a form of data integrity [4] expressed as labels on types in the language. Static analysis of information flow [47] 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 (§4–5):

- 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 (§8), 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.

2 Motivation

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 [39].

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 [12, 25, 45]. Latency to this single store is necessarily much higher for many users than latency to their own inboxes.

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 Figure 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 maintenance purposes, 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* [55], 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 Figure 1 does not address concurrent modification to the data. Suppose that a thread concurrently modifies `users` while another thread is executing Figure 1. Without care, this concurrent modification might invalidate Figure 1's iterator; at best, it is unclear whether 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 [51] or Salt [56], 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

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

Figure 1. Naive code for sending messages. Corrected MixT code is found in Figure 4.

```
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 2. Contest logic inside the mail delivery transaction. Corrected MixT code is found in Figure 13.

level. If we had a causal store such as TaRDiS [13] 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.

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.

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 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 Figure 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 6, MixT also allows intentional weakening of this restriction.

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.

3.1 MixT Language Syntax

Figure 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

$$
\begin{array}{rl}
x \in \textbf{Var} & f \in \textbf{Operation} \\
\oplus \in \textbf{Binop} & \ominus \in \textbf{Unop}
\end{array}
$$

$$
\begin{array}{rl}
(\mathit{Location})\ m & ::= x \mid *\,e \mid e.x \mid e\,\texttt{->}\,x \\
(\mathit{Expr})\ e & ::= m \mid e_1 \oplus e_2 \mid \ominus\, e \mid e_0.f(e_1, \ldots, e_n) \\
(\mathit{Stmt})\ s & ::= \texttt{var}\ x\ \texttt{=}\ e \mid m\ \texttt{=}\ e \mid \texttt{return}\ e \\
& \mid \texttt{while}\ \texttt{(}e\texttt{)}\ s \mid \texttt{if}\ \texttt{(}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \mid \texttt{\{}s_1, \ldots, s_n\texttt{\}}
\end{array}
$$

Figure 3. MixT surface syntax. Certain built-in operations are omitted for clarity of exposition.

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 Figure 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 *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.

3.2 A MixT Example Program

As an example of MixT code used within a larger program, the message delivery transaction of Section 2.2 is shown in Figure 4. To distinguish MixT code from surrounding C++ code, MixT code is colored green, whereas C++ code is blue.

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.

[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, …, e_n are passed as values by default, except when the store requests otherwise, at which point they will be dereferenced (resulting in a runtime error if these arguments refer to handles on the wrong store).

```
1 class user {
2   Handle<set<string>, causal, supports<insert>> inbox;
3 };

5 class group {
6   RemoteList<Handle<user, causal>, linearizable> users;
7   Handle<Log, eventual, supports<append>> log;
8   mixt_method(add_post) (post) mixt_captures(users,log) (
9     var iterator = users,
10    while (iterator.isValid()) {
11      log.append(iterator->v.inbox.insert(post)),
12      iterator = iterator->next
13    }
14  )
15 };
```

Figure 4. MixT message delivery implementation (§4.2). MixT code (lines 8–14) is colored green; C++ code is blue.

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).

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 Figure 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; Figure 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

```
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 5. Handle and DataStore interfaces.

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

Figure 6. Implementing a causal store in a host C++ program.

Handle. The only requirement from the DataStore API is the method beginTransaction(), which must create a new transaction 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.

Figure 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 (Figure 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 Figure 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.

4 Mixed-Consistency Transactions

Section 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 (§4.1). MixT's transaction support can provide atomic execution for Section 2.1's message delivery transaction (§4.5), and its type system will detect the fundamental errors of Section 2.3's contest (§4.3). A more detailed look at MixT's transactions comes in Section 5.

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 [25], 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 4.1 (Trace projection)**.**

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

Definition 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 [51], there is always some minimum consistency model onto which all events can be projected.

Definition 4.2 can also be adapted to transaction isolation levels [3] by considering traces containing explicit events that begin and end transactions. A full formalization is found in the technical report [44].

4.2 Noninterference for Mixed Consistency

In Section 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 [21, 47]. 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 correcntess condition normally associated with information flow (Section 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 [41].

Possibilistic security has been shown to be problematic in its original setting of confidentiality, because information can be leaked via refinement [52, 60]. In the context of consistency and other integrity-like properties, it does not seem to be a major concern [38].

4.3 Consistency as Information Flow

To enforce generalized noninterference, we treat consistency as a form of information-flow integrity [4] and use an information-flow type system [47] to outlaw bad programs. Previous work [52] 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 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 [47]. The type system of MixT (Section 5.2) statically catches invalid flows, whether implicit or explicit, and rejects unsafe transactions.

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 Figure 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 2.2.

MixT implements mixed-consistency transactions like this one by automatically splitting their code into a single subtransaction 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 2.2. This splitting does not automatically preserve atomicity, the subject of Section 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 [10, 61]. 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 5.3.

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

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

Figure 7. MixT flattened syntax.

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 7.1) can reduce its overhead. The performance evaluation (Section 8) shows that a complex MixT program can achieve reasonable throughput even when witnesses are used. We revisit witnesses in more detail in Section 5.4; formal arguments regarding the correctness of witnesses can be found in the technical report [44].

5 Formalizing the MixT Language

5.1 Desugared Language

To facilitate transaction splitting, MixT's surface syntax is translated to a "flattened" language whose syntax appears in Figure 7. There are a few notable changes from the surface language. All expressions are flattened by the compiler using standard techniques [48]. 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.

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 [47].

Figure 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

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell_1 \qquad \Gamma \vdash e : \tau \qquad \Delta, x_V : \ell \mid \Gamma, x_V : \tau \mid pc \vdash s : \ell_2 \qquad \ell_1 \sqsubseteq \ell \qquad x \notin \Gamma \qquad x \notin \Delta}{\Delta \mid \Gamma \mid pc \vdash \texttt{var}\ x\ \texttt{=}\ e\ \texttt{in}\ s : \ell}$$

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

REMOTE-READ

$$\frac{pc \sqsubseteq \ell}{\Delta, x_R : \ell \mid \Gamma \mid pc \vdash x : \ell} \qquad \Delta, x_V : \ell \mid \Gamma \mid pc \vdash x : \ell$$

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

Figure 8. Selected consistency typing rules for MixT. The labeled REMOTE-READ rule is unusual.

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 ($\perp$) 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.

Omitted from Figure 8 are rules governing assignment, operations (treated identically to assignments), and implicit flows; these follow standard conventions for static information flow [47] and can be found in the technical report [44]. Also omitted is the rule governing endorsement, discussed in Section 6.

5.3 Transaction Splitting

Figure 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.

$$\begin{aligned} (\mathit{Expr})\ e &::= \dots \mid \texttt{rand()} \mid \texttt{peek}(x) \\ (\mathit{Stmt})\ s &::= \dots \mid \texttt{release\_all}(n) \\ &\mid \texttt{acquire}(x, n, \ell_1, \dots, \ell_n) \\ &\mid \texttt{advance}(x) \mid \texttt{advance remote}(x) \end{aligned}$$

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

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

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

$$[\![\texttt{while}\ (e)\ \mathit{stmt} : \ell]\!]_\ell \triangleq \texttt{while}\ (e)\ [\![\mathit{stmt}]\!]_\ell$$

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

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

Figure 9. Selected transaction splitting rules.

```
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 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

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

Figure 11. Enhanced DataStore interface for witnesses.

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 Figure 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 Figure 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.

5.4 Atomicity for Split Transactions

A user may choose to request full atomicity for transactions that write to multiple stores. In order to ensure these writes are fully atomic, MixT employs the witness mechanism described in Section 4.5. Write witnesses correspond to lock-acquire events, while commit witnesses correspond to lock-release events. Both are represented by objects written directly to underlying storage. Write witnesses contain the locations of all corresponding commit witnesses and a list of all consistency levels involved in the transaction. Commit witnesses are empty objects. We extend the requirements on underlying storage to include a key–value API through which witnesses can be named (Figure 11). 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.

After transaction splitting, the MixT compiler augments the split transaction with code to create witnesses. First, the transaction generates a new variable `wit`, which we will use as the name of our commit witness. To avoid collisions, this name is selected randomly from a reserved 63-bit namespace. Next, it inserts the statement `acquire(`x`,wit,`*phases*`)` before all mutative operations. This statement creates a write witness, writing it alongside x. At the end of the phase, the compiler appends `release_all(wit)`, which writes the commit witness itself.

The reasoning behind this 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.

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 to some transaction which wrote this value. 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 in the worklist, MixT polls p's store until the commit witness becomes available, and then removes it from the worklist. As described in Section 7.2, 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 7.1.

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 Figure 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.

5.5 Satisfying Mixed Consistency

Now that we have reviewed the mechanisms of MixT in detail, we return to the question of *mixed consistency* itself; how do we know that MixT transactions satisfy definition 4.2?

If the consistency model, as is traditional, includes only information about independent reads and writes made to

```
    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 12. Strongly consistent contest logic with endorsement.

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 ($\downharpoonright$) also preserves all matching writes in that trace. In this sort of consistency model, the projection operator simply selects sets of memory locations.

6 Upgrading Consistency

6.1 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 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 Figure 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 (Figure 12). This code is correct, but runs all operations at a strong consistency level—exactly what we were trying to avoid. To improve performance, we can guard each strongly consistent read with a preliminary weakly consistent test, restoring

```
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 13. 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`.

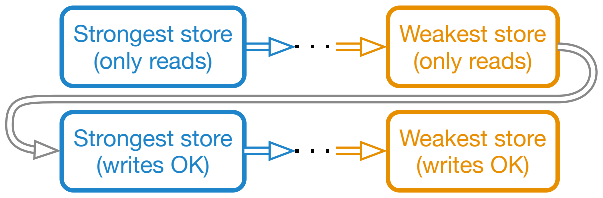
Strongest store (only reads)
Weakest store (only reads)
Strongest store (writes OK)
Weakest store (writes OK)

Figure 14. Transaction phases with endorsement.

causal execution for most transactions while ensuring the declaration of a winner is still guarded by a strongly consistent condition (Figure 13). 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.

6.2 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.

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. 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 14. If full isolation is required, read witnesses may be employed. Read witnesses are dual to write witnesses; the creation of a read witness accompanies every read, and checking for read witnesses occurs before every write. More details on read witnesses can be found in the technical report [44]. An optimization which uses read validation in place of read witnesses (as in optimistic concurrency control) is also possible [34].

7 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 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.)

7.1 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 [2, 13, 39, 40, 46]. 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.

7.2 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 [17, 18, 30, 53]. 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.

7.3 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[2]. Our entire compiler is written in constexpr C++ [43], allowing it to run during the "template expansion" step of compilation of the surrounding C++ code. Specifically, `mixt_`$\langle$*keyword*$\rangle$ 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 is deferred until runtime. 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

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

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.

8 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?

8.1 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 ± 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. Our causal and linearizable stores are both backed by PostgreSQL. To implement causal consistency, we created four replicas of data within the database, each associated with a four-entry vector clock version number. More details on the experimental setup are available in the technical report [44].

8.2 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 2.2. This benchmark features four more-complex transactions: message delivery (Figure 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.

8.3 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 a speedup compared to the simple alternative of running transactions entirely with linearizability. Figure 15 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

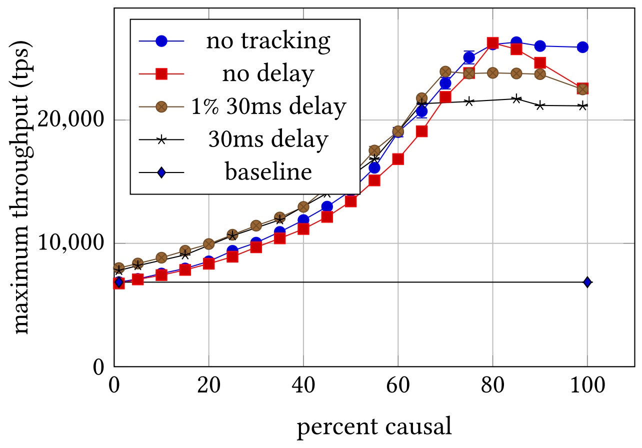

Figure 15. Maximum throughput as a function of linearizable mix for a 70% read workload. The blue (top circle) series shows maximum achievable throughput in transactions per second (tps) without witnesses; the remaining series shows full witness tracking with progressive artificial latency. The solid black line marks 0% causal without tracking (also the leftmost blue point), which serves as a baseline.

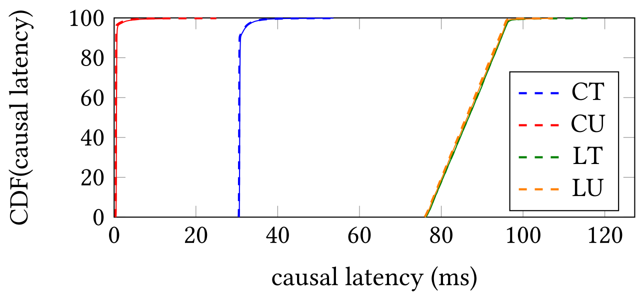

Figure 16. CDF plots for operation latency. C: Causal, L: Linearizable, T: Tracked, U: Untracked. Dashed lines: reads, solid lines: writes. All linearizable lines appear atop each other on the right.

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, discussed in our technical report [44]. 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 Figure 15, 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.

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	✓	✓

Figure 17. 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.

Latency Witnesses do affect latency, especially because of design decisions made for backward compatibility. Figure 16 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.

8.4 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 all 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 (Figure 17). 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 ± 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 ± 15 tps). As expected, mixed-consistency transactions yield significant speedup.

9 Related Work

Quelea [51] and Disciplined Inconsistency [26] 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.

Choosing Consistency Levels Choosing appropriate consistency models for data is a problem orthogonal to our work. Prior work [22, 24, 26, 27, 36, 37, 51] provide languages of constraints to describe data invariants, in turn providing the weakest consistency possible while still satisfying those constraints. Other work [7, 22, 29] 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 [15] and distributed systems [1, 16, 28, 40, 42, 50] 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 [11, 14, 35, 37, 46], but without providing any semantic guarantees across operations. Others provide tools to tune consistency based primarily on performance considerations [9, 54, 58]. Guerraoui et al. [23] 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 [19, 32, 33, 37, 58] 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 [7, 29, 56], but do not handle the semantic anomalies involved. A few systems [15, 58] 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 [20]. Some prior systems do enable programmers to mix transactions of different consistency with strong guarantees [37, 49, 57]. 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.

Enhancing Consistency of Existing Systems Beyond SQL, some existing work has focused on mechanisms that upgrade the consistency guarantees of weakly consistent underlying stores [2, 28, 50]. Indeed, several projects [40, 51] use this approach internally, adding consistency layers atop existing distributed systems like Cassandra.

10 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.

Acknowledgments

The authors would like to thank Josh Brown, Stephanie DeMane, Jon Sailor, Cody Mello, and Robert Mustacchi for their help determining industry applications. The authors would like to thank Ethan Cecchetti for help with PostgreSQL, Nate Nystrom for discussions about information flow and consistency, Isaac Sheff for extensive consultation, and Tom Magrino, Daoji Huang, Eric Lee, Eston Schweickart, Andrew Hirsch, and Sydney Zink for their editorial assistance.

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 grant 1717554.

References

[1] Masoud Saeida Ardekani, 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, 163–172.

[2] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-On Causal Consistency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 761–772.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1–10.

[4] K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report ESD-TR-76-372. USAF Electronic Systems Division, Bedford, MA. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[5] Eric Brewer. 2010. A Certain Freedom: Thoughts on the CAP Theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 335–335.

[6] Joshua William Skilken Brown. 2017. Personal communication. (Nov. 2017). Google, Inc.

[7] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *44th ACM Symp. on Principles of Programming Languages (POPL)*. 458–472.

[8] Josiah L. Carlson. 2013. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA.

[9] Shankha Chatterjee and Wojciech Golab. 2017. Self-Tuning Eventually-Consistent Data Stores. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 78–92.

[10] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. 2012. Automatic Partitioning of Database Applications. *PVLDB* 5, 11 (Aug. 2012), 1471–1482.

[11] Brian F Cooper, 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. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.

[12] James C Corbett, 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. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[13] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1615–1628.

[14] Giuseppe DeCandia, 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)*. 205–220.

[15] Brijesh Dongol, Radha Jagadeesan, and James Riely. 2018. Transactions in Relaxed Memory Architectures. In *45th ACM Symp. on Principles of Programming Languages (POPL)*. 18:1–18:29.

[16] Jiaqing Du, 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, 1–13.

[17] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *Proceedings of the VLDB Endowment* 9, 11 (2016), 852–863.

[18] Jennie Duggan, 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. *ACM SIGMOD Record* 44, 2 (2015), 11–16.

[19] Lei Gao, 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, 449–460.

[20] Mimi Gentz, 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. (2017).

[21] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symp. on Security and Privacy*. 11–20.

[22] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *43rd ACM Symp. on Principles of Programming Languages (POPL)*. 371–384.

[23] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 169–184.

[24] M. Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages* 1, 13 (Jan 1991), 124–149.

[25] M. Herlihy and J. Wing. 1988. *Linearizability: A Correctness Condition for Concurrent Objects*. Technical Report CMU-CS-88-120. Carnegie Mellon University, Pittsburgh, Pa.

[26] Brandon Holt, 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, 279–293.

[27] Brandon Holt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2015. Claret: Using Data Types for Highly Concurrent Distributed Transactions. In *1st Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*. Article 4, 4:1–4:4 pages.

[28] Ta-Yuan Hsu and Ajay D. Kshemkalyani. 2018. Value the Recent Past: Approximate Causal Consistency for Partially Replicated Systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 1 (2018), 212–225.

[29] Gowtham Kaki, Kartik Nagar, Mahsa Nazafzadeh, and Suresh Jagannathan. 2017. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *arXiv preprint arXiv:1710.09844* (2017).

[30] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB* 1, 2 (Aug. 2008), 1496–1499.

[31] Rusty Klophaus. 2010. Riak Core: Building Distributed Applications Without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*. ACM, New York, NY, USA, Article 14, 1 pages.

[32] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay Only When it Matters. *Proceedings of the VLDB Endowment* 2, 1 (2009), 253–264.

[33] Tim Kraska, 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, 113–126.

[34] H. T. Kung and J. T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. on Database Systems* 6, 2 (June 1981), 213–226.

[35] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[36] Cheng Li, Joao Leitao, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *USENIX Annual Technical Conference*.

[37] Cheng Li, 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)*.

[38] Jed Liu and Andrew C. Myers. 2014. Defining and Enforcing Referential Security. In *3rd Conf. on Principles of Security and Trust (POST)*. 199–219.

[39] Wyatt Lloyd, 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)*. 401–416.

[40] Wyatt Lloyd, 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)*. 313–328.

[41] Daryl McCullough. 1987. Specifications for Multi-Level Security and a Hook-up Property. In *IEEE Symp. on Security and Privacy*. IEEE Press, 161–161.

[42] Syed Akbar Mehdi, 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*. 453–468.

[43] Scott Meyers. 2014. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++ 11 and C++ 14*. O'Reilly Media, Inc..

[44] Matthew Milano and Andrew Myers. 2017. MixT: A Language for Mixing Consistency in Geodistributed Transactions: Technical Report. (2017).

[45] C. H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653.

[46] Eelco Plugge, Peter Membrey, and Tim Hawkins. 2010. *The Definitive Guide to MongoDB: The noSQL Database for Cloud and Desktop Computing*. Apress.

[47] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

[48] Amr Sabry and Matthias Felleisen. 1993. Reasoning About Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3–4 (Nov. 1993), 289–360.

[49] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. on Database Systems* 20, 3 (Sept. 1995), 325–363.

[50] Kazuyuki Shudo and Takashi Yaguchi. 2017. Causal Consistency for Data Stores and Applications as They are. *Journal of Information Processing* 25 (2017), 775–782.

[51] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 413–424.

[52] Geoffrey Smith and Dennis Volpano. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *25th ACM Symp. on Principles of Programming Languages (POPL)*. 355–364.

[53] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[54] Douglas B. Terry, 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)*. 309–324.

[55] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15th ACM Symp. on Operating System Principles (SOSP)*. 172–183.

[56] Chao Xie, 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. 495–509.

[57] Chao Xie, 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)*. ACM, New York, NY, USA, 279–294.

[58] Yingyi Yang, 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, 183–190.

[59] Haifeng Yu 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)*. Article 21, 14 pages.

[60] Steve Zdancewic and Andrew C. Myers. 2003. Observational Determinism for Concurrent Program Security. In *16th IEEE Computer Security Foundations Workshop (CSFW)*. 29–43.

[61] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. on Computer Systems* 20, 3 (Aug. 2002), 283–328.