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

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 runtime 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.

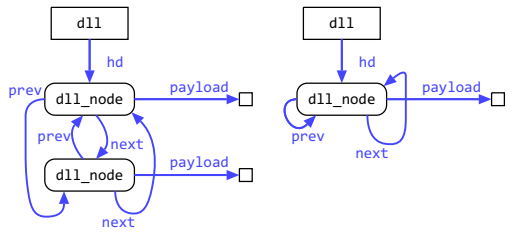

Figure 3. Two circular doubly linked lists, of size 2 and of size 1.

```
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.

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

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.

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

E1 - Evaluation-Context-Step

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

E2 - Variable-Ref-Step

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

E5A - Final-Reference-Step–Variable

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

E7A - Final-Assignment-Step–Variable

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

E8 - Assign-Var-Step

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

E11 - Declare-Var-Step

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

E15A - If-Disconnected-Success-Step

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

E15B - If-Disconnected-Failure-Step

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

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

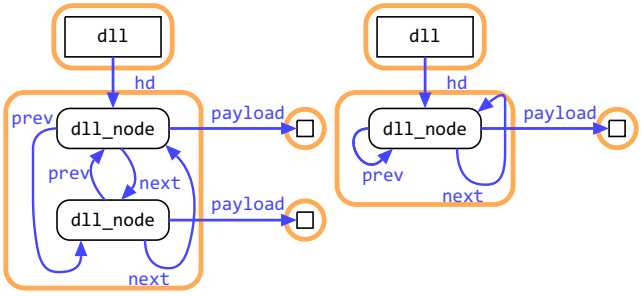

Figure 8. Two circular doubly linked lists, with regions drawn.

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

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

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

T5 - Isolated-Field-Reference
$$\frac{\texttt{iso}\ f\ \tau_f \in fields(\tau) \qquad r^{\circ}\langle x^{\circ'}[f \rightarrowtail r_f, F], X\rangle \in \mathcal{H} \qquad r_f^{\circ''}\langle X'\rangle \in \mathcal{H}}{\mathcal{H};x : r\ \tau,\Gamma \vdash x.f : r_f\ \tau_f \dashv \mathcal{H};x : r\ \tau,\Gamma}$$

T6 - Non-Isolated-Field-Assignment
$$\frac{\mathcal{H};\Gamma \vdash e_f : r\ \tau_f \dashv \mathcal{H}';\Gamma' \qquad \mathcal{H}';\Gamma' \vdash e : r\ \tau \dashv \mathcal{H}'';\Gamma'' \qquad \cdot f\ \tau_f \in fields(\tau)}{\mathcal{H};\Gamma \vdash e.f = e_f : r\ \tau_f \dashv \mathcal{H}'';\Gamma''}$$

T7 - Isolated-Field-Assignment
$$\frac{\mathcal{H};\Gamma \vdash e_f : r_f\ \tau_f \dashv \mathcal{H}', r^{\circ}\langle x^{\circ'}[f \rightarrowtail r_{old}, F], X\rangle; x : r\ \tau,\Gamma' \qquad \texttt{iso}\ f\ \tau_f \in fields(\tau)}{\mathcal{H};\Gamma \vdash x.f = e_f : r_f\ \tau_f \dashv \mathcal{H}', r^{\circ}\langle x^{\circ'}[f \rightarrowtail r_f, F], X\rangle; x : r\ \tau,\Gamma'}$$

T8 - Assign-Var
$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma', x : r_{old}\ \tau \qquad x \notin vars(\mathcal{H}')}{\mathcal{H};\Gamma \vdash x = e : r\ \tau \dashv \mathcal{H}';\Gamma', x : r\ \tau}$$

T10 - New-Loc
$$\mathcal{H};\Gamma \vdash \texttt{new-}\tau : r\ \tau \dashv \mathcal{H}, r^{\cdot}\langle\rangle;\Gamma$$

T11 - Declare-Var
$$\frac{\mathcal{H};\Gamma, x : \bot\ \tau \vdash e : r\ \tau' \dashv \mathcal{H}';\Gamma', x : r_{out}\ \tau \qquad x \notin vars(\mathcal{H}')}{\mathcal{H};\Gamma \vdash \texttt{declare}\ x : \tau\ \texttt{in}\ \{e\} : r\ \tau' \dashv \mathcal{H}';\Gamma'}$$

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

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 `new-`$\tau()$'s derivation has the form $r^{\cdot}\langle\rangle; x : r\ \tau$, but the field reference rule requires a context like $r^{\cdot}\langle x^{\cdot}[f \rightarrowtail r_f]\rangle, r_f^{\cdot}\langle\rangle; x : r\ \tau$.

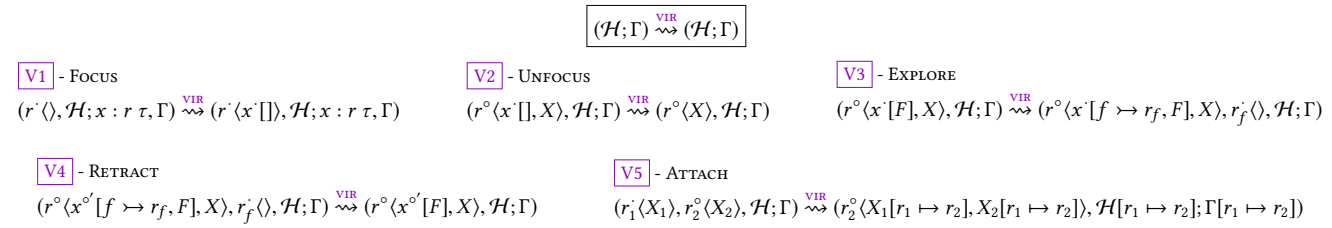

Figure 11. Virtual Transformation Rules.

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.

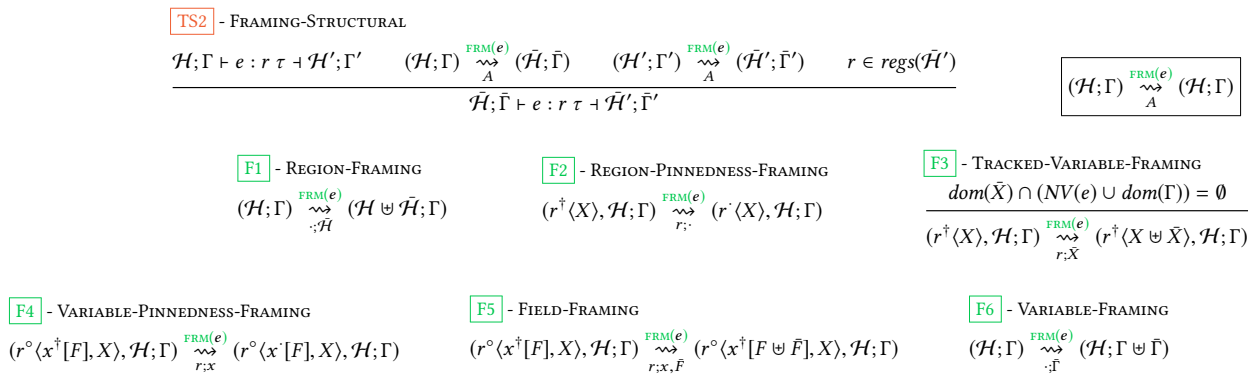

Figure 12. Framing rules

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

$$\frac{\mathcal{H};\Gamma \vdash e : r\ \tau \dashv \mathcal{H}';\Gamma' \qquad \tau_{fn} = (\mathcal{H};\Gamma) \Rightarrow (\mathcal{H}';\Gamma';r,\tau) \qquad (fn,\tau f) \in \mathcal{F}}{\vdash \mathsf{def}\ fn : \tau f\{e\}}\ \text{T0 - Function-Definition}$$

$$\frac{\overline{\Gamma \vdash x_i : r_i\ \tau_i} \qquad \overline{r_i' \mapsto r_i} \sqsubseteq \Phi_r \in \mathit{bijections}(\mathit{RegionNames}) \qquad \overline{(fn,(\mathcal{H};x_1' : r_1'\ \tau_1,\ldots,x_n' : r_n'\ \tau_n) \Rightarrow (\mathcal{H}';\Gamma';r_0',\tau_0)) \in \mathcal{F}} \qquad \Phi_x = \overline{x_i' \mapsto x_i} \in \mathit{bijections}(\mathit{VariableNames})}{\Phi_x(\Phi_r(\mathcal{H}));\Gamma \vdash fn(x_1,\ldots,x_n) : r_0\ \tau_0 \dashv \Phi_x(\Phi_r(\mathcal{H}'));\Phi_x(\Phi_r(\Gamma'))}\ \text{T9 - Function-Application}$$

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 nth node of a doubly linked list

to the nth 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 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 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

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

EC3 - Communication-Paired-Step

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

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.